# Lesson 4

Video / Lesson Forum

Welcome to Lesson 4! We are going to finish our journey through these key applications. We've already looked at a range of vision applications. We've looked a classification, localization, image regression. We briefly touched on NLP. We're going to do a deeper dive into NLP transfer learning today. We're going to then look at tabular data and collaborative filtering which are both super useful applications.

Then we're going to take a complete u-turn. We're going to take that collaborative filtering example and dive deeply into it to understand exactly what's happening mathematically?exactly what's happening in the computer. And we're going to use that to gradually go back in reverse order through the applications again in order to understand exactly what's going on behind the scenes of all of those applications.

### Correction on CamVid result

Before we do, somebody on the forum is kind enough to point out that when we compared ourselves to what we think might be the state of the art or was recently the state of the art for CamVid, there wasn't a fair comparison because the paper actually used a small subset of the classes, and we used all of the classes. So Jason in our study group was kind enough to rerun the experiments with the correct subset of classes from the paper, and our accuracy went up to 94% compared to 91.5% of the paper. So I think that's a really cool result. and a great example of how pretty much just using the defaults nowadays can get you far beyond what was the best of a year or two ago. It was certainly the best last year when we were doing this course because we started it quite intensely. So that's really exciting.

## Natural Language Processing (NLP) [2:00]

What I wanted to start with is going back over NLP a little bit to understand really what was going on there.

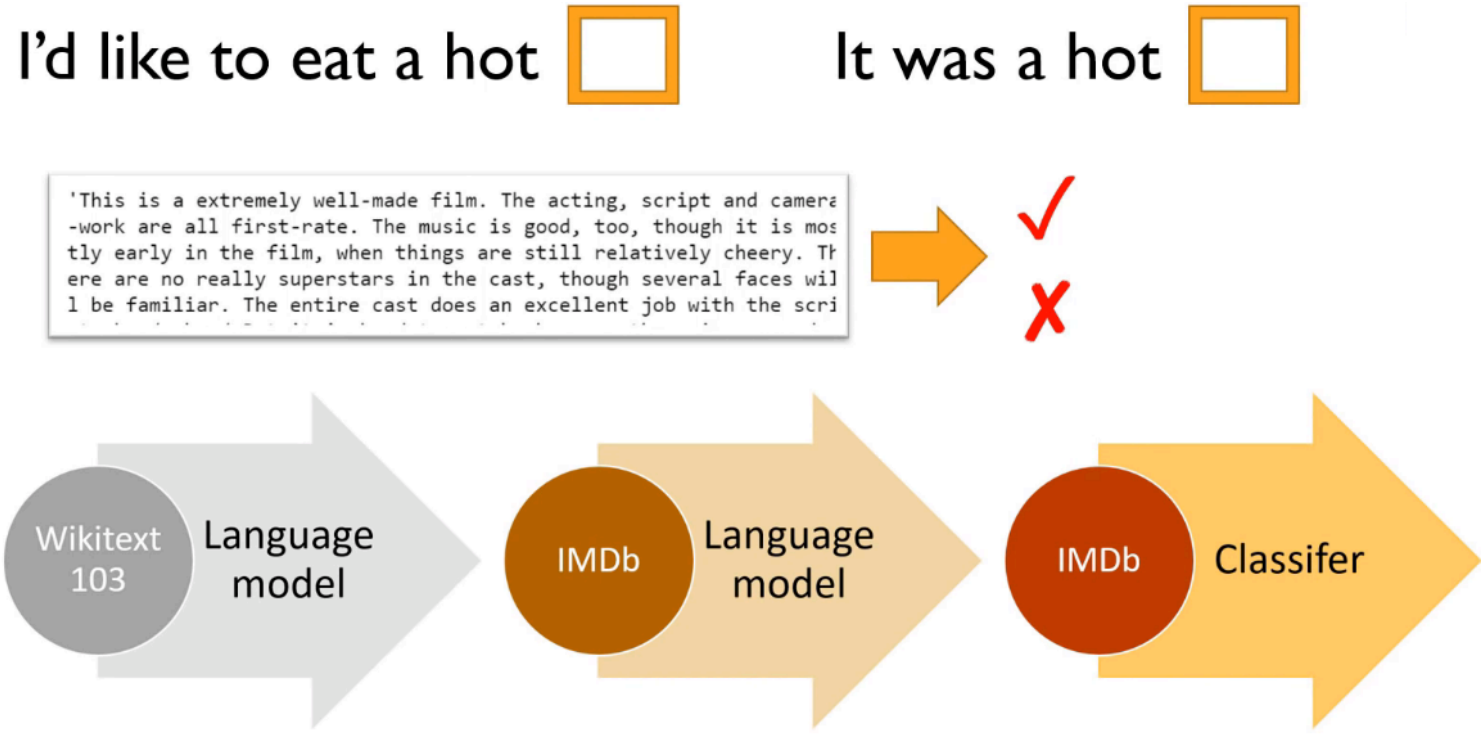
### A quick review

So first of all, a quick review. Remember NLP is natural language processing. It's about taking text and doing something with it. Text classification is particularly useful?practically useful applications. It's what we're going to start off focusing on. Because classifying a text or classifying a document can be used for anything from:

- Spam prevention
- Identifying fake news
- Finding a diagnosis from medical reports
- Finding mentions of your product in Twitter

So it's pretty interesting. And actually there was a great example during the week from one of our students [@howkhang](https://forums.fast.ai/u/howkhang) who is a lawyer and he mentioned on the forum that he had a really great results from classifying legal texts using this NLP approach. And I thought this was a great example. This is the post that they presented at an academic conference this week describing the approach:

This series of three steps that you see here (and I'm sure you recognize this classification matrix) is what we're going to start by digging into.




We're going to start out with a movie review like this one and decide whether it's positive or negative sentiment about the movie. That is the problem. We have, in the training set, 25,000 movie reviews and for each one we have like one bit of information: they liked it, or they didn't like it. That's what we're going to look into a lot more detail today and in the current lessons. Our neural networks (remember, they're just a bunch of matrix multiplies and simple nonlinearities?particularly replacing negatives with zeros), those weight matrices start out random. So if you start out with with some random parameters and try to train those parameters to learn how to recognize positive vs. negative movie reviews, you literally have 25,000 ones and zeros to actually tell you I like this one I don't like that one. That's clearly not enough information to learn, basically, how to speak English?how to speak English well enough to recognize they liked this or they didn't like this. Sometimes that can be pretty nuanced. Particularly with movie reviews because these are like online movie reviews on IMDB, people can often use sarcasm. It could be really quite tricky.

Until very recently, in fact, this year, neural nets didn't do a good job at all of this kind of classification problem. And that was why?there's not enough information available. So the trick, hopefully you can all guess, is to use transfer learning. It's always the trick.

Last year in this course I tried something crazy which was I thought what if I try transform learning to demonstrate that it can work for NLP as well. I tried it out and it worked extraordinarily well. So here we are, a year later, and transfer learning in NLP is absolutely the hit thing. And I'm going to describe to you what happens.

## Transfer learning in NLP [6:04]

The key thing is we're going to start with the same kind of thing that we used for computer vision?a pre-trained model that's been trained to do something different to what we're doing with it. For ImageNet, that was originally built as a model to predict which of a thousand categories each photo falls into. And people then fine-tune that for all kinds of different things as you've seen. So we're going to start with a pre-trained model that's going to do something else. Not movie review classification. We're going to start with a pre-trained model which is called a language model.

A language model has a very specific meaning in NLP and it's this. A language model is a model that learns to

predict the next word of a sentence. To predict the next word of a sentence, you actually have to know quite a lot about English (assuming you're doing it in English) and quite a lot of world knowledge. By world knowledge, I'll give you an example.

Here's your language model and it has read:

- "I'd like to eat a hot ___": Obviously,"dog", right?
- "It was a hot ___": Probably"day"

Now previous approaches to NLP use something called n-grams largely which is basically saying how often do these pairs or triplets of words tend to appear next to each other. And n-grams are terrible at this kind of thing. As you can see, there's not enough information here to decide what the next word probably is. But with a neural net, you absolutely can.

So here's the nice thing. If you train a neural net to predict the next word of a sentence then you actually have a lot of information. Rather than having a single bit to every 2,000 word movie review: "liked it" or "didn't like it", every single word, you can try and predict the next word. So in a 2,000 word movie review, there are 1,999 opportunities to predict the next word. Better still, you don't just have to look at movie reviews. Because really the hard thing isn't so much as "does this person like the movie or not?" but "how do you speak English?". So you can learn "how do you speak English?" (roughly) from some much bigger set of documents. So what we did was we started with Wikipedia.

**Wikitext 103 [8:30]**

Stephen Merity and some of his colleagues built something called Wikitext 103 dataset which is simply a subset of most of the largest articles from Wikipedia with a little bit of pre-processing that's available for download. So you're basically grabbing Wikipedia and then I built a language model on all of Wikipedia. So I just built a neural net which would predict the next word in every significantly sized Wikipedia article. That's a lot of information. If I remember correctly, it's something like a billion tokens. So we've got a billion separate things to predict. Every time we make a mistake on one of those predictions, we get the loss, we get gradients from that, and we can update our weights, and they can better and better until we can get pretty good at predicting the next word of Wikipedia.

Why is that useful? Because at that point, I've got a model that knows probably how to complete sentences like this, so it knows quite a lot about English and quite a lot about how the world works?what kinds of things tend to be hot in different situations, for instance. Ideally, it would learn things like "in 1996 in a speech to the United Nations, United States president _____ said"… Now that would be a really good language model, because it would actually have to know who is this United States president in that year. So getting really good at training language models is a great way to teach a neural-net a lot about what is our world, what's in our world, how do things work in our world. It's a really fascinating topic, and it's actually one that philosophers have been studying for hundreds of years now. There's actually a whole theory of philosophy which is about what can be learned from studying language alone. So it turns out, apparently, quite a lot.

So here's the interesting thing. You can start by training a language model on all of Wikipedia, and then we can make that available to all of you. Just like a pre-trained ImageNet model for vision, we've now made available a pre-trained Wikitext model for NLP not because it's particularly useful of itself (predicting the next word of sentences is somewhat useful, but not normally what we want to do), but it's a model that understands a lot about language and a lot about what language describes. So then, we can take that and we can do transfer learning to create a new language model that's specifically good at predicting the next word of movie reviews.

**Fine-tuning Wikitext to create a new language model [11:10]**

If we can build a language model that's good at predicting the next word of movie reviews pre-trained with the Wikitext model, then that's going to understand a lot about "my favorite actor is Tom ____." Or "I thought the photography was fantastic but I wasn't really so happy about the _____ (director)." It's going to learn a lot about specifically how movie reviews are written. It'll even learn things like what are the names of some popular movies.

That would then mean we can still use a huge corpus of lots of movie reviews even if we don't know whether they're positive or negative to learn a lot about how movie reviews are written. So for all of this pre-training and all of this language model fine-tuning, we don't need any labels at all. It is what the researcher Yann LeCun calls **self supervised learning**. In other words, it's a classic supervised model?we have labels, but the labels are not things that somebody else have created. They're built into the dataset itself. So this is really really neat. Because at this point, we've now got something that's good at understanding movie reviews and we can fine-tune that with transfer learning to do the thing we want to do which in this case is to classify movie reviews to be positive or negative. So my hope was (when I tried this last year) that at that point, 25,000 ones and zeros would be enough feedback to fine-tune that model and it turned out it absolutely was.

**Question**: Does the language model approach works for text in forums that are informal English, misspelled words or slangs or shortforms like s6 instead of Samsung S 6? [12:47]

Yes, absolutely it does. Particularly if you start with your wikitext model and then fine-tune it with your "target" corpus. Corpus is just a bunch of documents (emails, tweets, medical reports, or whatever). You could fine-tune it so it can learn a bit about the specifics of the slang , abbreviations, or whatever that didn't appear in the full corpus. So interestingly, this is one of the big things that people were surprised about when we did this research last year. People thought that learning from something like Wikipedia wouldn't be that helpful because it's not that representative of how people tend to write. But it turns out it's extremely helpful because there's a much a difference between Wikipedia and random words than there is between like Wikipedia and reddit. So it kind of gets you 99% of the way there.

So language models themselves can be quite powerful. For example there was a blog post from SwiftKey (the folks that do the mobile-phone predictive text keyboard) and they describe how they kind of rewrote their underlying model to use neural nets. This was a year or two ago. Now most phone keyboards seem to do this. You'll be typing away on your mobile phone, and in the prediction there will be something telling you what word you might want next. So that's a language model in your phone.

Another example was the researcher Andrej Karpathy who now runs all this stuff at Tesla, back when he was a PhD student, he created a language model of text in LaTeX documents and created these automatic generation of LaTeX documents that then became these automatically generated papers. That's pretty cute.

We're not really that interested in the output of the language model ourselves. We're just interested in it because it's helpful with this process.

**Review of the basic process [15:14]**

We briefly looked at the process last week. The basic process is, we're going to start with the data in some format. So for example, we've prepared a little IMDB sample that you can use which is in CSV file. You can read it in with Pandas and there's negative or positive, the text of each movie review, and boolean of is it in the validation set or the training set.

```
path = untar_data(URLs.IMDB_SAMPLE)
path.ls()
```

```
[PosixPath('/home/jhoward/.fastai/data/imdb_sample/texts.csv'),
 PosixPath('/home/jhoward/.fastai/data/imdb_sample/models')]
```

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

| | label | text | is_valid |
|---|---|---|---|
| 0 | negative | Un-bleeping-believable! Meg Ryan doesn't even … | False |
| 1 | positive | This is a extremely well-made film. The acting… | False |
| 2 | negative | Every once in a long while a movie will come a… | False |
| 3 | positive | Name just says it all. I watched this movie wi… | False |
| 4 | negative | This movie succeeds at being one of the most u… | False |

So there's an example of a movie review:

```
df['text'][1]
```

```
'This is a extremely well-made film. The acting, script and camera-work are all f
```

So you can just go `TextDataBunch.from_csv` to grab a language model specific data bunch:

```
data_lm = TextDataBunch.from_csv(path, 'texts.csv')
```

And then you can create a learner from that in the usual way and fit it.

```
data_lm.save()
```

You can save the data bunch which means that the pre-processing that is done, you don't have to do it again. You can just load it.

```
data = TextDataBunch.load(path)
```

What happens behind the scenes if we now load it as a classification data bunch (that's going to allow us to see the labels as well)?

```
data = TextClasDataBunch.load(path)
data.show_batch()
```

| text | label |
|---|---|
| xxbos xxfld 1 raising victor vargas : a review you know , raising victor vargas is like sticking your hands into a big , xxunk bowl of xxunk . it 's warm and gooey , but you 're not sure if it feels right . try as i might | negative |
| xxbos xxfld 1 now that che(2008 ) has finished its relatively short australian cinema run ( extremely limited xxunk screen in xxunk , after xxunk ) , i can xxunk join both xxunk of " at the movies " in taking steven soderbergh to task . it 's usually | negative |
| xxbos xxfld 1 many xxunk that this is n't just a classic due to the fact that it 's the first xxup 3d game , or even the first xxunk - up . it 's also one of the first xxunk games , one of the xxunk definitely the first | positive |
| xxbos xxfld 1 i really wanted to love this show . i truly , honestly did . for the first time , gay viewers get their own version of the " the bachelor " . with the help of his obligatory " hag " xxunk , james , a | negative |
| xxbos xxfld 1 this film sat on my xxunk for weeks before i watched it . i xxunk a self - indulgent xxunk | positive |

| text | label |
| --- | --- |
| flick about relationships gone bad . i was wrong ; this was an xxunk xxunk into the xxunk - up xxunk of new xxunk . the | |

As we described, it basically creates a separate unit (i.e. a "token") for each separate part of a word. So most of them are just for words, but sometimes if it's like an `'s` from `it's`, it will get its own token. Every bit of punctuation tends to get its own token (a comma, a full stop, and so forth).

Then the next thing that we do is a numericalization which is where we find what are all of the unique tokens that appear here, and we create a big list of them. Here's the first ten in order of frequency:

```
data.vocab.itos[:10]
```

```
['xxunk', 'xxpad', 'the', ',', '.', 'and', 'a', 'of', 'to', 'is']
```

And that big list of unique possible tokens is called the vocabulary which we just call it a "vocab". So what we then do is we replace the tokens with the ID of where is that token in the vocab:

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

```
Text xxbos xxfld 1 he now has a name , an identity , some memories and a a lost g
```

```
data.train_ds[0][0].data[:10]
```

```
array([ 43,  44,  40,  34, 171,  62,   6, 352,   3,  47])
```

That's numericalization. Here's the thing though. As you'll learn, every word in our vocab is going to require a separate row in a weight matrix in our neural net. So to avoid that weight matrix getting too huge, we restrict the vocab to no more than (by default) 60,000 words. And if a word doesn't appear more than two times, we don't put it in the vocab either. So we keep the vocab to a reasonable size in that way. When you see these `xxunk`, that's an unknown token. It just means this was something that was not a common enough word to appear in our vocab.

We also have a couple of other special tokens like (see `fastai.text.transform.py` for up-to-date info):

- `xxfld`: This is a special thing where if you've got like title, summary, abstract, body, (i. e. separate parts of a document), each one will get a separate field and so they will get numbered (e.g. `xxfld 2`).
- `xxup`: If there's something in all caps, it gets lower cased and a token called `xxup` will get added to it.

**With the data block API [18:31]**

Personally, I more often use the data block API because there's less to remember about exactly what data bunch to use, and what parameters and so forth, and it can be a bit more flexible.

```
data = (TextList.from_csv(path, 'texts.csv', cols='text')
                .split_from_df(col=2)
                .label_from_df(cols=0)
                .databunch())
```

So another approach to doing this is to just decide:

- What kind of list you're creating (i.e. what's your independent variable)? So in this case, my independent

variable is text.
- What is it coming from? A CSV.
- How do you want to split it into validation versus training? So in this case, column number two was the `is_valid` flag.
- How do you want to label it? With positive or negative sentiment, for example. So column zero had that.
- Then turn that into a data bunch.

That's going to do the same thing.

```
path = untar_data(URLs.IMDB)
path.ls()
```

```
[PosixPath('/home/jhoward/.fastai/data/imdb/imdb.vocab'),
 PosixPath('/home/jhoward/.fastai/data/imdb/models'),
 PosixPath('/home/jhoward/.fastai/data/imdb/tmp_lm'),
 PosixPath('/home/jhoward/.fastai/data/imdb/train'),
 PosixPath('/home/jhoward/.fastai/data/imdb/test'),
 PosixPath('/home/jhoward/.fastai/data/imdb/README'),
 PosixPath('/home/jhoward/.fastai/data/imdb/tmp_clas')]
```

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

```
[PosixPath('/home/jhoward/.fastai/data/imdb/train/pos'),
 PosixPath('/home/jhoward/.fastai/data/imdb/train/unsup'),
 PosixPath('/home/jhoward/.fastai/data/imdb/train/unsupBow.feat'),
 PosixPath('/home/jhoward/.fastai/data/imdb/train/labeledBow.feat'),
 PosixPath('/home/jhoward/.fastai/data/imdb/train/neg')]
```

Now let's grab the whole data set which has:

- 25,000 reviews in training
- 25,000 interviews in validation
- 50,000 unsupervised movie reviews (50,000 movie reviews that haven't been scored at all)

**Language model [19:44]**

We're going to start with the language model. Now the good news is, we don't have to train the Wikitext 103 language model. Not that it's difficult?you can just download the wikitext 103 corpus, and run the same code. But it takes two or three days on a decent GPU, so not much point in you doing it. You may as well start with ours. Even if you've got a big corpus of like medical documents or legal documents, you should still start with Wikitext 103. There's just no reason to start with random weights. It's always good to use transfer learning if you can.

So we're gonna start fine-tuning our IMDB language model.

```
bs=48
```

```
data_lm = (TextList.from_folder(path)
           #Inputs: all the text files in path
            .filter_by_folder(include=['train', 'test'])
           #We may have other temp folders that contain text files so we only kee
            .random_split_by_pct(0.1)
```

```
            #We randomly split and keep 10% (10,000 reviews) for validation
             .label_for_lm()
            #We want to do a language model so we label accordingly
             .databunch(bs=bs))
data_lm.save('tmp_lm')
```

We can say:

- It's a list of text files?the full IMDB actually is not in a CSV. Each document is a separate text file.
- Say where it is?in this case we have to make sure we just to include the `train` and `test` folders.
- We randomly split it by 0.1.

Now this is interesting?10%. Why are we randomly splitting it by 10% rather than using the predefined train and test they gave us? This is one of the cool things about transfer learning. Even though our validation set has to be held aside, it's actually only the labels that we have to keep aside. So we're not allowed to use the labels in the test set. If you think about in a Kaggle competition, you certainly can't use the labels because they don't even give them to you. But you can certainly use the independent variables. So in this case, you could absolutely use the text that is in the test set to train your language model. This is a good trick?when you do the language model, concatenate the training and test set together, and then just split out a smaller validation set so you've got more data to train your language model. So that's a little trick.

So if you're doing NLP stuff on Kaggle, for example, or you've just got a smaller subset of labeled data, make sure that you use all of the text you have to train in your language model, because there's no reason not to.

- How are we going to label it? Remember, a language model kind of has its own labels. The text itself is labels so label for a language model (`label_for_lm`) does that for us.
- And create a data bunch and save it. That takes a few minutes to tokenize and numericalize.

Since it takes some few minutes, we save it. Later on you can just load it. No need to run it again.

```
data_lm = TextLMDataBunch.load(path, 'tmp_lm', bs=bs)

data_lm.show_batch()
```

| idx | text |
|---|---|
| 0 | xxbos after seeing the truman show , i wanted to see the other films by weir . i would say this is a good one to start with . the plot : the wife of a doctor ( who is trying to impress his bosses , so he can get xxunk trying to finish her written course , while he s at work . but one day a strange man , who says that he s a plumber , tells her he s been called out to repair some pipes in there flat . |
| 1 | and turn to the wisdom of homeless people & ghosts . that 's a good plan . i would never recommend this movie ; partly because the sexual content is unnecessarily graphic , but also because it really does n't offer any valuable insight . check out " yentl " if you want to see a much more useful treatment of jewish tradition at odds with society . xxbos creep is the story of kate ( potente ) , an intensely unlikeable bourgeois bitch that finds herself somehow sleeping through the noise of the last |
| 2 | been done before but there is something about the way its done here that lifts it up from the rest of the pack . 8 out of 10 for dinosaur / monster lovers . xxbos i rented this movie to see how the sony xxunk camera shoots , ( i recently purchased the same camera ) and was blown away by the story and the acting . the directing , acting , editing was all above what i expected from what appeared at first glance to be a " low budget " type of |
| 3 | troubles . nigel and xxunk are the perfect team , i 'd watch their show any day ! i was so crushed when they removed it , and anytime they had it on xxup tv after that i was over the moon ! they put it on on demand one |

| idx | text |
|---|---|
| 4 | summer ( only the first eight episodes or so ) and i 'd spend whole afternoons watching them one after the other … but the worst part ? it is now back on a channel called xxunk - and xxup it xxup 's xxup on movie ! ) the movie is about edward , a obsessive - compulsive , nice guy , who happens to be a film editor . he is then lent to another department in the building , and he is sent to the posh yet violent world of sam campbell , the splatter and gore department . sam campbell , eddy 's new boss , is telling eddy about the big break on his movies , the gruesome loose limbs series , and he needs eddy to make the movie somewhat less violent so they can |

**Training [22:29]**

At this point things are going to look very familiar. We create a learner:

```
learn = language_model_learner(data_lm, pretrained_model=URLs.WT103, drop_mult=0.
```

But instead of creating a CNN learner, we're going to create a language model learner. So behind the scenes, this is actually not going to create a CNN (a convolutional neural network), it's going to create an RNN (a recurrent neural network). We're going to be learning exactly how they're built over the coming lessons, but in short they're the same basic structure. The input goes into a weight matrix (i.e. a matrix multiply), that then you replace the negatives with zeros, and it goes into another matrix multiply, and so forth a bunch of times. So it's the same basic structure.

As usual, when we create a learner, you have to pass in two things:

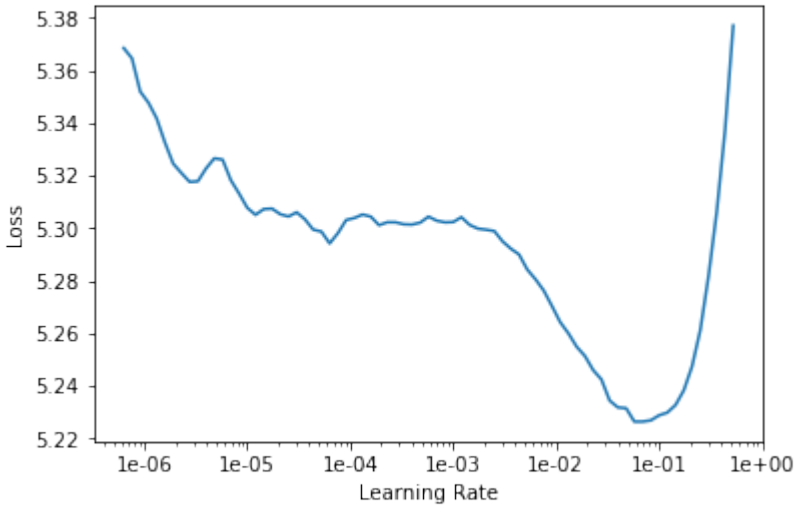
- The data: so here's our language model data
- What pre-trade model we want to use: here, the pre-trade model is the Wikitext 103 model that will be downloaded for you from fastai if you haven't used it before just like ImageNet pre-trained models are downloaded for you.

This here (`drop_mult=0.3`) sets the amount of dropout. We haven't talked about that yet. We've talked briefly about this idea that there is something called regularization and you can reduce the regularization to avoid underfitting. So for now, just know that by using a number lower than one is because when I first tried to run this, I was under fitting. So if you reduced that number, then it will avoid under fitting.

Okay. so we've got a learner, we can `lr_find` and looks pretty standard:

```
learn.lr_find()

learn.recorder.plot(skip_end=15)
```




Then we can fit one cycle.

```
learn.fit_one_cycle(1, 1e-2, moms=(0.8,0.7))

Total time: 12:42
epoch  train_loss  valid_loss  accuracy
1      4.591534    4.429290    0.251909  (12:42)
```

What's happening here is we are just fine-tuning the last layers. Normally after we fine-tune the last layers, the next thing we do is we go `unfreeze` and train the whole thing. So here it is:

```
learn.unfreeze()
learn.fit_one_cycle(10, 1e-3, moms=(0.8,0.7))

Total time: 2:22:17
epoch  train_loss  valid_loss  accuracy
1      4.307920    4.245430    0.271067  (14:14)
2      4.253745    4.162714    0.281017  (14:13)
3      4.166390    4.114120    0.287092  (14:14)
4      4.099329    4.068735    0.292060  (14:10)
5      4.048801    4.035339    0.295645  (14:12)
6      3.980410    4.009860    0.298551  (14:12)
7      3.947437    3.991286    0.300850  (14:14)
8      3.897383    3.977569    0.302463  (14:15)
9      3.866736    3.972447    0.303147  (14:14)
10     3.847952    3.972852    0.303105  (14:15)
```

As you can see, even on a pretty beefy GPU that takes two or three hours. In fact, I'm still under fitting. So probably tonight, I might train it overnight and try and do a little bit better. I'm guessing I could probably train this a bit longer because you can see the accuracy hasn't started going down again. So I wouldn't mind try to train that a bit longer. But the accuracy, it's interesting. 0.3 means we're guessing the next word of the movie review correctly about a third of the time. That sounds like a pretty high number?the idea that you can actually guess the next word that often. So it's a good sign that my language model is doing pretty well. For more limited domain documents (like medical transcripts and legal transcripts), you'll often find this accuracy gets a lot higher. So sometimes this can be even 50% or more. But 0.3 or more is pretty good.

**Predicting with Language Model [25:43]**

You can now run `learn.predict` and pass in the start of a sentence, and it will try and finish off that sentence for you.

```
learn.predict('I liked this movie because ', 100, temperature=1.1, min_p=0.001)
```

```
Total time: 00:10
```

```
'I liked this movie because  of course after yeah funny later that the world reas
```

Now I should mention, this is not designed to be a good text generation system. This is really more designed to check that it seems to be creating something that's vaguely sensible. There's a lot lot of tricks that you can use to generate much higher quality text?none of which we're using here. But you can kind of see that it's certainly not random words that it's generating. It sounds vaguely English like even though it doesn't make any sense.

At this point, we have a movie review model. So now we're going to save that in order to load it into our classifier (i.e. to be a pre-trained model for the classifier). But I actually don't want to save the whole thing. A lot of the second half of the language model is all about predicting the next word rather than about understanding the sentence so far. So the bit which is specifically about understanding the sentence so far is called the **encoder**, so I just save that (i.e. the bit that understands the sentence rather than the bit that generates the word).

```
learn.save_encoder('fine_tuned_enc')
```

**Classifier [27:18]**

Now we're ready to create our classifier. Step one, as per usual, is to create a data bunch, and we're going to do basically exactly the same thing:

```
data_clas = (TextList.from_folder(path, vocab=data_lm.vocab)
             #grab all the text files in path
             .split_by_folder(valid='test')
             #split by train and valid folder (that only keeps 'train' and 'test'
             .label_from_folder(classes=['neg', 'pos'])
             #remove docs with labels not in above list (i.e. 'unsup')
             .filter_missing_y()
             #label them all with their folders
             .databunch(bs=50))
data_clas.save('tmp_clas')
```

But we want to make sure that it uses exactly the same vocab that are used for the language model. If word number 10 was `the` in the language model, we need to make sure that word number 10 is `the` in the classifier. Because otherwise, the pre-trained model is going to be totally meaningless. So that's why we pass in the vocab from the language model to make sure that this data bunch is going to have exactly the same vocab. That's an important step.

`split_by_folder`?remember, the last time we had split randomly, but this time we need to make sure that the labels of the test set are not touched. So we split by folder.

And then this time we label it not for a language model but we label these classes (`['neg', 'pos']`). Then finally create a data bunch.

Sometimes you'll find that you ran out of GPU memory. I was running this in an 11G machine, so you should make sure this number (`bs`) is a bit lower if you run out of memory. You may also want to make sure you restart the notebook and kind of start it just from here (classifier section). Batch size 50 is as high as I could get on an 11G card. If you're using a p2 or p3 on Amazon or the K80 on Google, for example, I think you'll get 16G so you might be able to make this bit higher, get it up to 64. So you can find whatever batch size fits on your card.

So here is our data bunch:

```
data_clas = TextClasDataBunch.load(path, 'tmp_clas', bs=bs)
data_clas.show_batch()
```

| text | label |
|---|---|
| xxfld 1 match 1 : tag team table match bubba ray and spike dudley vs eddie guerrero and chris benoit bubba ray and spike dudley started things off with a tag team table match against eddie guerrero and chris benoit . according to the rules of the match , both | pos |
| xxfld 1 i have never seen any of spike lee 's prior films , as their trailers never caught my interest . i have seen , and admire denzel washington , and jodie foster 's work , and have several of their dvds . i was , however , entirely | neg |
| xxfld 1 pier paolo pasolini , or pee - pee - pee as i prefer to call him ( due to his love of showing male genitals ) , is perhaps xxup the most overrated european marxist director - and they are thick on the ground . how anyone can | neg |
| xxfld 1 chris rock deserves better than he gives himself in " down to earth . " as directed by brothers chris & paul weitz of " american pie " fame , this uninspired remake of warren beatty 's 1978 fantasy " heaven can wait , " itself a rehash | neg |
| xxfld 1 yesterday , i went to the monthly antique flea market that comes to town . i really have no interest in such things , but i went for the fellowship of friends who do have such an interest . looking over the hundreds of vendor , passing many | pos |

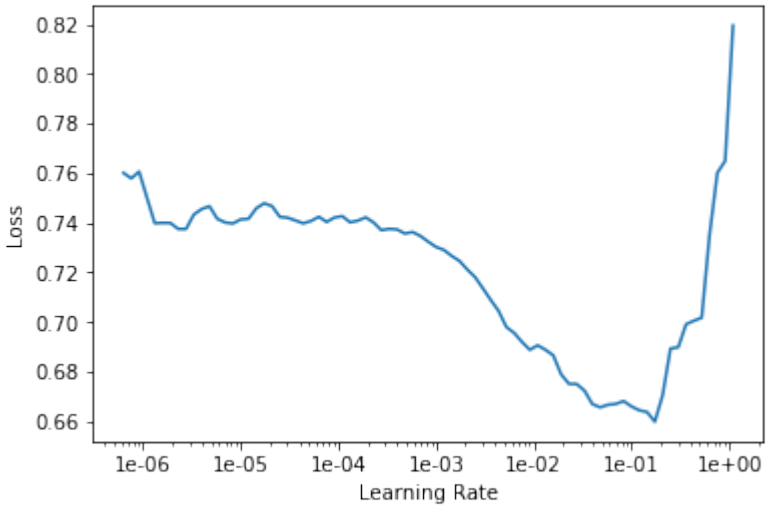
```
learn = text_classifier_learner(data_clas, drop_mult=0.5)
learn.load_encoder('fine_tuned_enc')
learn.freeze()
```

This time, rather than creating a language model learner, we're creating a text classifier learner. But again, same thing?pass in the data that we want, figure out how much regularization we need. If you're overfitting then you can increase this number (`drop_mult`). If you're underfitting, you can decrease the number. And most importantly, load in our pre train model. Remember, specifically it's this half of the model called the encoder which is the bit that we want to load in.

Then freeze, lr_find, find the learning rate and fit for a little bit.

```
learn.lr_find()
```

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




```
learn.fit_one_cycle(1, 2e-2, moms=(0.8,0.7))
```

```
Total time: 02:46
epoch  train_loss  valid_loss  accuracy
1      0.294225    0.210385    0.918960   (02:46)
```

We're already up nearly to 92% accuracy after less than three minutes of training. So this is a nice thing. In your particular domain (whether it be law, medicine, journalism, government, or whatever), you probably only need to train your domain's language model once. And that might take overnight to train well. But once you've got it, you can now very quickly create all kinds of different classifiers and models with that. In this case, already a pretty good model after three minutes. So when you first start doing this. you might find it a bit annoying that your first models take four hours or more to create that language model. But the key thing to remember is you only have to do that once for your entire domain of stuff that you're interested in. And then you can build lots of different classifiers and other models on top of that in a few minutes.

```
learn.save('first')
```

```
learn.load('first');
```

```
learn.freeze_to(-2)
learn.fit_one_cycle(1, slice(1e-2/(2.6**4),1e-2), moms=(0.8,0.7))
```

```
Total time: 03:03
epoch  train_loss  valid_loss  accuracy
1      0.268781    0.180993    0.930760   (03:03)
```

We can save that to make sure we don't have to run it again.

And then, here's something interesting. I'm not going to say `unfreeze`. Instead, I'm going to say `freeze_to`. What that says is unfreeze the last two layers, don't unfreeze the whole thing. We've just found it really helps with these text classification not to unfreeze the whole thing, but to unfreeze one layer at a time.

- unfreeze the last two layers
- train it a little bit more
- unfreeze the next layer again
- train it a little bit more

- unfreeze the whole thing
- train it a little bit more

```
learn.save('second')

learn.load('second');

learn.freeze_to(-3)
learn.fit_one_cycle(1, slice(5e-3/(2.6**4),5e-3), moms=(0.8,0.7))

Total time: 04:06
epoch  train_loss  valid_loss  accuracy
1      0.211133    0.161494    0.941280  (04:06)

learn.save('third')

learn.load('third');

learn.unfreeze()
learn.fit_one_cycle(2, slice(1e-3/(2.6**4),1e-3), moms=(0.8,0.7))

Total time: 10:01
epoch  train_loss  valid_loss  accuracy
1      0.188145    0.155038    0.942480  (05:00)
2      0.159475    0.153531    0.944040  (05:01)
```

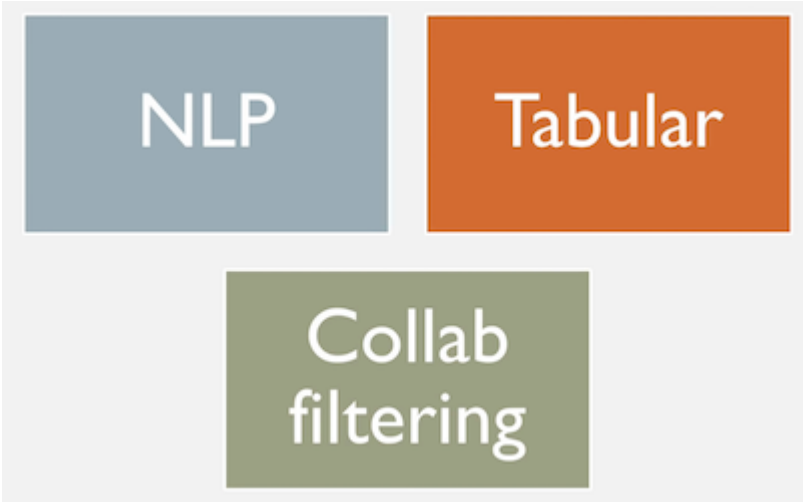
You also see I'm passing in this thing (`moms=(0.8,0.7)`)?momentums equals 0.8,0.7. We are going to learn exactly what that means probably next week. We may even automate it. So maybe by the time you watch the video of this, this won't even be necessary anymore. Basically we found for training recurrent neural networks (RNNs), it really helps to decrease the momentum a little bit. So that's what that is.

That gets us a 94.4% accuracy after about half an hour or less of training. There's quite a lot less of training the actual classifier. We can actually get this quite a bit better with a few tricks. I don't know if we'll learn all the tricks this part. It might be next part. But even this very simple standard approach is pretty great.

| | | |
|---|---|---|
| IMDb | *BCN+Char+CoVe [Ours]* | *91.8* |
| | SA-LSTM [Dai and Le, 2015] | 92.8 |
| | bmLSTM [Radford et al., 2017] | 92.9 |
| | TRNN [Dieng et al., 2016] | 93.8 |
| | oh-LSTM [Johnson and Zhang, 2016] | 94.1 |
| | **Virtual [Miyato et al., 2017]** | **94.1** |

If we compare it to last year's state of the art on IMDb, this is from The CoVe paper from McCann et al. at Salesforce Research. Their paper was 91.8% accurate. And the best paper they could find, they found a fairly domain-specific sentiment analysis paper from 2017, they've got 94.1%. And here, we've got 94.4%. And the best models I've been able to build aince have been about 95.1%. So if you're looking to do text classification, this really standardized transfer learning approach works super well.

## Tabular [33:10]




So that was NLP. We'll be learning more about NLP later in this course. But now, I wanted to switch over and look at tabular. Now tabular data is pretty interesting because it's the stuff that, for a lot of you, is actually what you use day-to-day at work in spreadsheets, in relational databases, etc.

**Question**: Where does the magic number of $2.6^4$ in the learning rate come from? [33:38]

```
learn.fit_one_cycle(2, slice(1e-3/(2.6**4),1e-3), moms=(0.8,0.7))
```

Good question. So the learning rate is various things divided by 2.6 to the fourth. The reason it's to the fourth, you will learn about at the end of today. So let's focus on the 2.6. Why 2.6? Basically, as we're going to see in more detail later today, this number, the difference between the bottom of the slice and the top of the slice is basically what's the difference between how quickly the lowest layer of the model learns versus the highest layer of the model learns. So this is called discriminative learning rates. So really the question is as you go from layer to layer, how much do I decrease the learning rate by? And we found out that for NLP RNNs, the answer is 2.6.

How do we find out that it's 2.6? I ran lots and lots of different models using lots of different sets of hyper parameters of various types (dropout, learning rates, and discriminative learning rate and so forth), and then I created something called a random forest which is a kind of model where I attempted to predict how accurate my NLP classifier would be based on the hyper parameters. And then I used random forest interpretation methods to basically figure out what the optimal parameter settings were, and I found out that the answer for this number was 2.6. So that's actually not something I've published or I don't think I've even talked about it before, so there's a new piece of information. Actually, a few months after I did this, Stephen Merity and somebody else did publish a paper describing a similar approach, so the basic idea may be out there already.

Some of that idea comes from a researcher named Frank Hutter and one of his collaborators. They did some interesting work showing how you can use random forests to actually find optimal hyperparameters. So it's kind of a neat trick. A lot of people are very interested in this thing called Auto ML which is this idea of like building models to figure out how to train your model. We're not big fans of it on the whole. But we do find that building models to better understand how your hyper parameters work, and then finding those rules of thumb like oh basically it can always be 2.6 quite helpful. So there's just something we've kind of been playing with.

## Back to Tabular [36:41]

Let's talk about tabular data. Tabular data such as you might see in a spreadsheet, a relational database, or financial

report, it can contain all kinds of different things. I tried to make a little list of some of the kinds of things that I've seen tabular data analysis used for:


Using neural nets for analyzing tabular data?when we first presented this, people were deeply skeptical. They thought it was a terrible idea to use neural nets to analyze tabular data, because everybody knows that you should use logistic regression, random forests, or gradient boosting machines (all of which have their place for certain types of things). But since that time, it's become clear that the commonly held wisdom is wrong. It's not true that neural nets are not useful for tabular data ,in fact they are extremely useful. We've shown this in quite a few of our courses, but what's really helped is that some really effective organizations have started publishing papers and posts describing how they've been using neural nets for analyzing tabular data.

One of the key things that comes up again and again is that although feature engineering doesn't go away, it certainly becomes simpler. So Pinterest, for example, replaced the gradient boosting machines that they were using to decide how to put stuff on their homepage with neural nets. And they presented at a conference this approach, and they described how it really made engineering a lot easier because a lot of the hand created features weren't necessary anymore. You still need some, but it was just simpler. So they ended up with something that was more accurate, but perhaps even more importantly, it required less maintenance. So I wouldn't say you it's the only tool that you need in your toolbox for analyzing tabular data. But where else, I used to use random forests 99% of the time when I was doing machine learning with tabular data, I now use neural nets 90% of the time. It's my standard first go-to approach now, and it tends to be pretty reliable and effective.

One of the things that's made it difficult is that until now there hasn't been an easy way to create and train tabular neural nets. Nobody has really made it available in a library. So we've actually just created `fastai.tabular` and I think this is pretty much the first time that's become really easy to use neural nets with tabular data. So let me show you how easy it is.

**Tabular examples [39:51]**

lesson4-tabular.ipynb

This is actually coming directly from the examples folder in the fastai repo. I haven't changed it at all. As per usual, as well as importing fastai, import your application?so in this case, it's tabular.

```
from fastai import *
from fastai.tabular import *
```

We assume that your data is in a Pandas DataFram. Pandas DataFrame is the standard format for tabular data in Python. There are lots of ways to get it in there, but probably the most common might be `pd.read_csv`. But whatever your data is in, you can probably get it into a Pandas data frame easily enough.

```
path = untar_data(URLs.ADULT_SAMPLE)
df = pd.read_csv(path/'adult.csv')
```

**Question**: What are the 10% of cases where you would not default to neural nets? [40:41]

Good question. I guess I still tend to give them a try. But yeah, I don't know. It's kind of like as you do things for a while, you start to get a sense of the areas where things don't quite work as well. I have to think about that during the week. I don't think I have a rule of thumb. But I would say, you may as well try both. I would say try a random forest and try a neural net. They're both pretty quick and easy to run, and see how it looks. If they're roughly similar, I might dig into each and see if I can make them better. But if the random forest is doing way better, I'd probably just stick with that. Use whatever works.

So we start with the data in a data frame, and so we've got an adult sample?it's a classic old dataset. It's a pretty small simple old dataset that's good for experimenting with. And it's a CSV file, so you can read it into a data frame with Pandas read CSV (`pd.read_csv`). If your data is in a relational database, Pandas can read from that. If it's in spark or Hadoop, Pandas can read from that. Pandas can read from most stuff that you can throw at it. So that's why we use it as a default starting point.

```
dep_var = '>=50k'
cat_names = ['workclass', 'education', 'marital-status', 'occupation', 'relations
cont_names = ['age', 'fnlwgt', 'education-num']
procs = [FillMissing, Categorify, Normalize]

test = TabularList.from_df(df.iloc[800:1000].copy(), path=path, cat_names=cat_nam

data = (TabularList.from_df(df, path=path, cat_names=cat_names, cont_names=cont_n
                            .split_by_idx(list(range(800,1000)))
                            .label_from_df(cols=dep_var)
                            .add_test(test, label=0)
                            .databunch())
```

As per usual, I think it's nice to use the data block API. So in this case, the list that we're trying to create is a tabular list and we're going to create it from a data frame. So you can tell it:

- What the data frame is.
- What the path that you're going to use to save models and intermediate steps is.
- Then you need to tell it what are your categorical variables and what are your continuous variables.

**Continuous vs. Categorical [43:07]**

We're going to be learning a lot more about what that means to the neural net next week, but for now the quick summary is this. Your independent variables are the things that you're using to make predictions with. So things like education, marital status, age, and so forth. Some of those variables like age are basically numbers. They could be any number. You could be 13.36 years old or 19.4 years old or whatever. Where else, things like marital status are options that can be selected from a discrete group: married, single, divorced, whatever. Sometimes those options might be quite a lot more, like occupation. There's a lot of possible occupations. And sometimes, they might be binary (i.e. true or false). But anything which you can select the answer from a small group of possibilities is called a **categorical variable**. So we're going to need to use a different approach in the neural net to modeling categorical variables to what we use for continuous variables. For categorical variables, we're going to be using something called **embeddings** which we'll be learning about later today. For continuous variables, they could just be sent into the neural net just like pixels in a neural net can. Because pixels in a neural net are already numbers; these continuous things are already numbers as well. So that's easy.

So that's why you have to tell the tabular list from data frame which ones are which. There are some other ways to do that by pre-processing them in Pandas to make things categorical variables, but it's kind of nice to have one API for doing everything; you don't have to think too much about it.

**Processor [45:04]**

Then we've got something which is a lot like transforms in computer vision. Transforms in computer vision do things like flip a photo on its axis, turn it a bit, brighten it, or normalize it. But for tabular data, instead of having transforms, we have things called processes. And they're nearly identical but the key difference, which is quite important, is that a processor is something that happens ahead of time. So we basically pre-process the data frame rather than doing it as we go. So transformations are really for data augmentation?we want to randomize it and do it differently each time. Or else, processes are the things that you want to do once, ahead of time.

```
procs = [FillMissing, Categorify, Normalize]
```

We have a number of processes in the fastai library. And the ones we're going to use this time are:

- `FillMissing`: Look for missing values and deal with them some way.
- `Categorify`: Find categorical variables and turn them into Pandas categories
- `Normalize` : Do a normalization ahead of time which is to take continuous variables and subtract their mean and divide by their standard deviation so they are zero-one variables.

The way we deal with missing data, we'll talk more about next week, but in short, we replace it with the median and add a new column which is a binary column of saying whether that was missing or not.

For all of these things, whatever you do to the training set, you need to do exactly the same thing to the validation set and the test set. So whatever you replaced your missing values with, you need to replace them with exactly the same thing in the validation set. So fastai handles all these details for you. They are the kinds of things that if you have to do it manually, if you like me, you'll screw it up lots of times until you finally get it right. So that's what these processes are here.

Then we're going to split into training versus validation sets. And in this case, we do it by providing a list of indexes so the indexes from 800 to a thousand. It's very common. I don't quite remember the details of this dataset, but it's very common for wanting to keep your validation sets to be contiguous groups of things. If they're map tiles, they should be the map tiles that are next to each other, if their time periods, they should be days that are next to each other, if they are video frames, they should be video frames next to each other. Because otherwise you're

kind of cheating. So it's often a good idea to use `split_by_idx` and to grab a range that's next to each other if your data has some kind of structure like that or find some other way to structure it in that way.

All right, so that's now given us a training and a validation set. We now need to add labels. In this case, the labels can come straight from the data frame we grabbed earlier, so we just have to tell it which column it is. So the dependent variable is whether they're making over $50,000 salary. That's the thing we're trying to predict.

We'll talk about test sets later, but in this case we can add a test set. And finally get our data bunch. S at that point, we have something that looks like this:

```
data.show_batch(rows=10)
```

| workclass | education | marital-status | occupation | relationship | race | education-num_na | age | fnlwgt | education-num | target |
|---|---|---|---|---|---|---|---|---|---|---|
| Private | Prof-school | Married-civ-spouse | Prof-specialty | Husband | White | False | 0.1036 | 0.9224 | 1.9245 | 1 |
| Self-emp-inc | Bachelors | Married-civ-spouse | Farming-fishing | Husband | White | False | 1.7161 | -1.2654 | 1.1422 | 1 |
| Private | HS-grad | Never-married | Adm-clerical | Other-relative | Black | False | -0.7760 | 1.1905 | -0.4224 | 0 |
| Private | 10th | Married-civ-spouse | Sales | Own-child | White | False | -1.5823 | -0.0268 | -1.5958 | 0 |
| Private | Some-college | Never-married | Handlers-cleaners | Own-child | White | False | -1.3624 | 0.0284 | -0.0312 | 0 |
| Private | Some-college | Married-civ-spouse | Prof-specialty | Husband | White | False | 0.3968 | 0.4367 | -0.0312 | 1 |
| ? | Some-college | Never-married | ? | Own-child | White | False | -1.4357 | -0.7295 | -0.0312 | 0 |
| Self-emp-not-inc | 5th-6th | Married-civ-spouse | Sales | Husband | White | False | 0.6166 | -0.6503 | -2.7692 | 1 |
| Private | Some-college | Married-civ-spouse | Sales | Husband | White | False | 1.5695 | -0.8876 | -0.0312 | 1 |
| Local-gov | Some-college | Never-married | Handlers-cleaners | Own-child | White | False | -0.6294 | -1.5422 | -0.0312 | 0 |

There is our data. Then to use it, it looks very familiar. You get a learner, in this case it's a tabular learner, passing in the data, some information about your architecture, and some metrics. And you then call fit.

```
learn = tabular_learner(data, layers=[200,100], metrics=accuracy)

learn.fit(1, 1e-2)

Total time: 00:03
epoch  train_loss  valid_loss  accuracy
1      0.362837    0.413169    0.785000  (00:03)
```

**Question**: How to combine NLP (tokenized) data with meta data (tabular data) with Fastai? For instance, for IMBb classification, how to use information like who the actors are, year made, genre, etc. [49:14]

Yeah, we're not quite up to that yet. So we need to learn a little bit more about how neural net architectures work

as well. But conceptually, it's kind of the same as the way we combine categorical variables and continuous variables. Basically in the neural network, you can have two different sets of inputs merging together into some layer. It could go into an early layer or into a later layer, it kind of depends. If it's like text and an image and some metadata, you probably want the text going into an RNN, the image going into a CNN, the metadata going into some kind of tabular model like this. And then you'd have them basically all concatenated together, and then go through some fully connected layers and train them end to end. We will probably largely get into that in part two. In fact we might entirely get into that in part two. I'm not sure if we have time to cover it in part one. But conceptually, it's a fairly simple extension of what we'll be learning in the next three weeks.

**Question**: Do you think that things like `scikit-learn` and `xgboost` will eventually become outdated? Will everyone will use deep learning tools in the future? Except for maybe small datasets?[50:36]

I have no idea. I'm not good at making predictions. I'm not a machine learning model. I mean `xgboost` is a really nice piece of software. There's quite a few really nice pieces of software for gradient boosting in particular. Actually, random forests in particular has some really nice features for interpretation which I'm sure we'll find similar versions for neural nets, but they don't necessarily exist yet. So I don't know. For now, they're both useful tools. `scikit-learn` is a library that's often used for pre-processing and running models. Again, it's hard to predict where things will end up. In some ways, it's more focused on some older approaches to modeling, but I don't know. They keep on adding new things, so we'll see. I keep trying to incorporate more scikit-learn stuff into fastai and then I keep finding ways I think I can do it better and I throw it away again, so that's why there's still no scikit-learn dependencies in fastai. I keep finding other ways to do stuff.

[52:12]

```
learn = tabular_learner(data, layers=[200,100], metrics=accuracy)
```

We're gonna learn what `layers=` means either towards the end of class today or the start of class next week, but this is where we're basically defining our architecture just like when we chose ResNet 34 or whatever for conv nets. We'll look at more about metrics in a moment, but just to remind you, metrics are just the things that get printed out. They don't change our model at all. So in this case, we're saying I want you to print out the accuracy to see how we're doing.

So that's how to do tabular. This is going to work really well because we're gonna hit our break soon. And the idea was that after three and a half lessons, we're going to hit the end of all of the quick overview of applications, and then I'm going to go down on the other side. I think we're going to be to the minute, we're going to hit it. Because the next one is collaborative filtering.

## Collaborative Filtering [53:08]

Collaborative filtering is where you have information about who bought what, or who liked what? it's basically something where you have something like a user, a reviewer, or whatever and information about what they've bought, what they've written about, or what they reviewed. So in the most basic version of collaborative filtering, you just have two columns: something like user ID and movie ID and that just says this user bought that movie. So for example, Amazon has a really big list of user IDs and product IDs like what did you buy. Then you can add additional information to that table such as oh, they left a review, what review did they give it? So it's now like user ID, movie ID, number of stars. You could add a timecode so this user bought this product at this time and gave it this review. But they are all basically the same structure.

There are two ways you could draw that collaborative filtering structure. One is a two-column approach where

you've got user and movie. And you've got user ID, movie ID?each pair basically describes that user watch that movie, possibly also number of stars (3, 4, etc). The other way you could write it would be you could have like all the users down here and all the movies along here. And then, you can look and find a particular cell in there to find out what could be the rating of that user for that movie, or there's just a `1` there if that user watched that movie, or whatever.


So there are two different ways of representing the same information. Conceptually, it's often easier to think of it this way (the matrix on the right), but most of the time you won't store it that way. Explicitly because most of the time, you'll have what's called a very sparse matrix which is to say most users haven't watched most movies or most customers haven't purchased most products. So if you store it as a matrix where every combination of customer and product is a separate cell in that matrix, it's going to be enormous. So you tend to store it like the left or you can store it as a matrix using some kind of special sparse matrix format. If that sounds interesting, you should check out Rachel's computational linear algebra course on fastai where we have lots and lots of information about sparse matrix storage approaches. For now though, we're just going to kind of keep it in this format on left hand side.

[56:38]

lesson4-collab.ipynb

For collaborative filtering, there's a really nice dataset called MovieLens created by GroupLens group and you can download various different sizes (20 million ratings, 100,000 ratings). We've actually created an extra small version for playing around with which is what we'll start with today. And then probably next week, we'll use the bigger version.

```
from fastai import *
from fastai.collab import *
from fastai.tabular import *
```

You can grab the small version using `URLs.ML_SAMPLE`:

```
user,item,title = 'userId','movieId','title'

path = untar_data(URLs.ML_SAMPLE)
path
```

```
PosixPath('/home/jhoward/.fastai/data/movie_lens_sample')

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

| | userId | movieId | rating | timestamp |
|---|---|---|---|---|
| 0 | 73 | 1097 | 4.0 | 1255504951 |
| 1 | 561 | 924 | 3.5 | 1172695223 |
| 2 | 157 | 260 | 3.5 | 1291598691 |
| 3 | 358 | 1210 | 5.0 | 957481884 |
| 4 | 130 | 316 | 2.0 | 1138999234 |

It's a CSV so you can read it with Pandas and here it is. It's basically a list of user IDs?we don't actually know anything about who these users are. There's some movie IDs. There is some information about what the movies are, but we won't look at that until next week. Then there's the rating and the timestamp. We're going to ignore the timestamp for now. So that's a subset of our data. `head` in Pandas is just the first rows.

So now that we've got a data frame, the nice thing about collaborative filtering is it's incredibly simple.

```
data = CollabDataBunch.from_df(ratings, seed=42)

y_range = [0,5.5]

learn = collab_learner(data, n_factors=50, y_range=y_range)
```

That's all the data that we need. So you can now go ahead and say get `collab_learner` and you can pass in the data bunch. The architecture, you have to tell it how many factors you want to use, and we're going to learn what that means after the break. And then something that could be helpful is to tell it what the range of scores are. We're going to see how that helps after the break as well. So in this case, the minimum score is 0, the maximum score is 5.

```
learn.fit_one_cycle(3, 5e-3)

Total time: 00:04
epoch  train_loss  valid_loss
1      1.600185    0.962681    (00:01)
2      0.851333    0.678732    (00:01)
3      0.660136    0.666290    (00:01)
```

Now that you've got a learner, you can go ahead and call `fit_one_cycle` and trains for a few epochs, and there it is. So at the end of it, you now have something where you can pick a user ID and a movie ID, and guess whether or not that user will like that movie.

**Cold start problem [58:55]**

This is obviously a super useful application that a lot of you are probably going to try during the week. In past classes, a lot of people have taken this collaborative filtering approach back to their workplaces and discovered that using it in practice is much more tricky than this. Because in practice, you have something called the cold start problem. So the cold start problem is that the time you particularly want to be good at recommending movies is when you have a new user, and the time you particularly care about recommending a movie is when it's a new

movie. But at that point, you don't have any data in your collaborative filtering system and it's really hard.

As I say this, we don't currently have anything built into fastai to handle the cold start problem and that's really because the cold start problem, the only way I know of to solve it (in fact, the only way I think that conceptually can solve it) is to have a second model which is not a collaborative filtering model but a metadata driven model for new users or new movies.

I don't know if Netflix still does this, but certainly what they used to do when I signed up to Netflix was they started showing me lots of movies and saying "have you seen this?" "did you like it?" ?so they fixed the cold start problem through the UX, so there was no cold stat problem. They found like 20 really common movies and asked me if I liked them, they used my replies to those 20 to show me 20 more that I might have seen, and by the time I had gone through 60, there was no cold start problem anymore.

For new movies, it's not really a problem because like the first hundred users who haven't seen the movie go in and say whether they liked it, and then the next hundred thousand, the next million, it's not a cold start problem anymore.

The other thing you can do if you, for whatever reason, can't go through that UX of asking people did you like those things (for example if you're selling products and you don't really want to show them a big selection of your products and say did you like this because you just want them to buy), you can instead try and use a metadata based tabular model what geography did they come from maybe you know their age and sex, you can try and make some guesses about the initial recommendations.

So collaborative filtering is specifically for once you have a bit of information about your users and movies or customers and products or whatever.

[1:01:37]

**Question**: How does the language model trained in this manner perform on code switched data (Hindi written in English words), or text with a lot of emojis?

Text with emojis, it'll be fine. There's not many emojis in Wikipedia and where they are at Wikipedia it's more like a Wikipedia page about the emoji rather than the emoji being used in a sensible place. But you can (and should) do this language model fine-tuning where you take a corpus of text where people are using emojis in usual ways, and so you fine-tune the Wikitext language model to your reddit or Twitter or whatever language model. And there aren't that many emojis if you think about it. There are hundreds of thousands of possible words that people can be using, but a small number of possible emojis. So it'll very quickly learn how those emojis are being used. So that's a piece of cake.

I'm not really familiar with Hindi, but I'll take an example I'm very familiar with which is Mandarin. In Mandarin, you could have a model that's trained with Chinese characters. There are about five or six thousand Chinese characters in common use, but there's also a romanization of those characters called pinyin. It's a bit tricky because although there's a nearly direct mapping from the character to the pinyin (I mean there is a direct mapping but that pronunciations are not exactly direct), there isn't direct mapping from the pinyin to the character because one pinyin corresponds to multiple characters.

So the first thing to note is that if you're going to use this approach for Chinese, you would need to start with a Chinese language model.

Actually fastai has something called Language Model Zoo where we're adding more and more language models for different languages, and also increasingly for different domain areas like English medical texts or even

language models for things other than NLP like genome sequences, molecular data, musical MIDI notes, and so forth. So you would you obviously start there.

To then convert that (in either simplified or traditional Chinese) into pinyin, you could either map the vocab directly, or as you'll learn, these multi-layer models?it's only the first layer that basically converts the tokens into a set of vectors, you can actually throw that away and fine-tune just the first layer of the model. So that second part is going to require a few more weeks of learning before you exactly understand how to do that and so forth, but if this is something you're interested in doing, we can talk about it on the forum because it's a nice test of understanding.

**Question**: What about time series on tabular data? is there any RNN model involved in `tabular.models`? [1:05:09]

We're going to look at time series tabular data next week, but the short answer is generally speaking you don't use a RNN for time series tabular data but instead, you extract a bunch of columns for things like day of week, is it a weekend, is it a holiday, was the store open, stuff like that. It turns out that adding those extra columns which you can do somewhat automatically basically gives you state-of-the-art results. There are some good uses of RNNs for time series, but not really for these kind of tabular style time series (like retail store logistics databases, etc).

**Question**: Is there a source to learn more about the cold start problem? [1:06:14]

I'm gonna have to look that up. If you know a good resource, please mention it on the forums.

**The half way point [1:06:34]**

That is both the break in the middle of lesson 4, it's the halfway point of the course, and it's the point at which we have now seen an example of all the key applications. So the rest of this course is going to be digging deeper into how they actually work behind the scenes, more of the theory, more of how the source code is written, and so forth. So it's a good time to have a nice break. Furthermore, it's my birthday today, so it's a really special moment.

## Collaborative filter with Microsoft Excel [1:07:25]

collab_filter.xlsx

| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | **movieId** | **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 | |
| 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 | 5 | 2 | 5 | 4 |
| 17 | | | | | | | | | | | | | | | | | | | | | | |
| 18 | | | | | | | | | | | | | | | | | | | | | | |
| 19 | | NB: These are initialized to random numbers | | | | | | -1.69 | 1.49 | -0.14 | 1.95 | -0.09 | 1.80 | 1.74 | 0.68 | 0.22 | 1.92 | 1.87 | 1.69 | -1.16 | 1.66 | 1.35 |
| 20 | | Then we use Solver to optimize them → | | | | | | 1.01 | 0.12 | 1.36 | 1.49 | 1.17 | 0.73 | -0.20 | -0.01 | 2.06 | 1.40 | 1.23 | 0.91 | 1.93 | 0.66 | 0.08 |
| 21 | | with gradient descent | | | | | | 0.82 | 1.48 | 0.02 | 0.53 | 1.07 | 1.24 | 1.64 | 0.95 | 0.43 | 0.82 | 0.42 | 0.71 | 0.99 | 0.57 | 1.47 |
| 22 | | | | ↓ | | | | 1.89 | 0.50 | 1.74 | 0.41 | 1.57 | 0.49 | 0.20 | 1.54 | 0.43 | -0.22 | 0.25 | 0.19 | 1.39 | 0.46 | 0.72 |
| 23 | | | | | | | **movieId** | 2.39 | 1.13 | 1.15 | -0.74 | 1.14 | -0.63 | 0.90 | 1.24 | 1.11 | 0.19 | 0.43 | 0.43 | 1.11 | 0.47 | 0.90 |
| 24 | | | | | | **userId** | | **27** | **49** | **57** | **72** | **79** | **89** | **92** | **99** | **143** | **179** | **180** | **197** | **402** | **417** | **505** |
| 25 | | 0.21 | 1.61 | 2.89 | -1.26 | 0.82 | **14** | 3.25 | 5.10 | 0.98 | 3.23 | 3.93 | 3.99 | 5.29 | 1.97 | 4.98 | 5.45 | 3.65 | 3.97 | 4.90 | 2.87 | 4.51 |
| 26 | | 1.55 | 0.75 | 0.22 | 1.62 | 1.26 | **29** | 4.40 | 4.98 | 5.08 | 3.99 | 4.95 | 3.61 | 4.37 | 5.32 | 4.08 | 4.10 | 4.88 | 4.31 | 3.53 | 4.55 | 4.79 |
| 27 | | 1.50 | 1.17 | 0.22 | 1.08 | 1.49 | **72** | 4.43 | 4.94 | 4.98 | 4.13 | 4.86 | 3.42 | 4.30 | 4.74 | 4.96 | 4.75 | 5.27 | 4.60 | 3.90 | 4.61 | 4.58 |
| 28 | | 0.47 | 0.89 | 1.32 | 1.13 | 0.77 | **211** | 5.16 | 4.21 | 4.01 | 2.83 | 5.06 | 3.19 | 3.74 | 4.27 | 3.84 | 0.00 | 3.15 | 3.08 | 4.91 | 3.01 | 0.00 |

Microsoft Excel is one of my favorite ways to explore data and understand models. I'll make sure I put this in the repo, and actually this one, we can probably largely do in Google sheets. I've tried to move as much as I can over the last few weeks into Google sheets, but I just keep finding this is such a terrible product, so please try to find a copy of Microsoft Excel because there's nothing close, I've tried everything. Anyway, spreadsheets get a bad rap from people that basically don't know how to use them. Just like people who spend their life on Excel and then they start using Python, and they're like what the heck is this stupid thing. It takes thousands of hours to get really good at spreadsheets, but a few dozen hours to get confident at them. Once you're confident at them, you can see everything in front of you. It's all laid out, it's really great.

## Jeremy's spreadsheet tip of the day! [1:08:37]

I'll give you one spreadsheet tip today which is if you hold down the `ctrl` key or `command` key on your keyboard and press the arrow keys, here's `ctrl+?`, it takes you to the end of a block of a table that you're in. And it's by far the best way to move around the place, so there you go.

In this case, I want to skip around through this table, so I can hit `ctrl+ ?? ?` to get to the bottom right, `ctrl+ ?? ??` to get to the top left. Skip around and see what's going on.

So here's some data, and as we talked about, one way to look at collaborative filtering data is like this:

| F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | movieId | 27 | 49 | 57 | 72 | 79 | 89 | 92 | 99 | 143 | 179 | 180 | 197 | 402 | 417 | 505 |
| userId | 14 | 3 | 5 | 1 | 3 | 4 | 4 | 5 | 2 | 5 | 5 | 4 | 5 | 5 | 2 | 5 |
| | 29 | 5 | 5 | 5 | 4 | 5 | 4 | 4 | 5 | 4 | 4 | 5 | 5 | 3 | 4 | 5 |
| | 72 | 4 | 5 | 5 | 4 | 5 | 3 | 4.5 | 5 | 4.5 | 5 | 5 | 5 | 4.5 | 5 | 4 |
| | 211 | 5 | 4 | 4 | 3 | 5 | 3 | 4 | 4.5 | 4 | | 3 | 3 | 5 | 3 | |
| | 212 | 2.5 | | 2 | 5 | | 4 | 2.5 | | 5 | 5 | 3 | 3 | 4 | 3 | 2 |
| | 293 | 3 | | 4 | 4 | 4 | 3 | | 3 | 4 | 4 | 4.5 | 4 | 4.5 | 4 | |
| | 310 | 3 | 3 | 5 | 4.5 | 5 | 4.5 | 2 | 4.5 | 4 | 3 | 4.5 | 4.5 | 4 | 3 | 4 |
| | 379 | 5 | 5 | 5 | 4 | | 4 | 5 | 4 | 4 | 4 | | 3 | 5 | 4 | 4 |
| | 451 | 4 | 5 | 4 | 5 | 4 | 4 | 5 | 5 | 4 | 4 | 4 | 4 | 2 | 3.5 | 5 |
| | 467 | 3 | 3.5 | 3 | 2.5 | | | 3 | 3.5 | 3.5 | 3 | 3.5 | 3 | 3 | 4 | 4 |
| | 508 | 5 | 5 | 4 | 3 | 5 | 2 | 4 | 4 | 5 | 5 | 5 | 3 | 4.5 | 3 | 4.5 |
| | 546 | | 5 | 2 | 3 | 5 | | 5 | 5 | | 2.5 | 2 | 3.5 | 3.5 | 3.5 | 5 |
| | 563 | 1 | 5 | 3 | 5 | 4 | 5 | 5 | | 2 | 5 | 5 | 3 | 3 | 4 | 5 |
| | 579 | 4.5 | 4.5 | 3.5 | 3 | 4 | 4.5 | 4 | 4 | 4 | 4 | 3.5 | 3 | 4.5 | 4 | 4.5 |
| | 623 | | 5 | 3 | 3 | | 3 | 5 | | 5 | 5 | 5 | 5 | 2 | 5 | 4 |

What we did was we grabbed from the MovieLens data the people that watched the most movies and the movies that were the most watched, and just limited the dataset down to those 15. As you can see, when you do it that way, it's not sparse anymore. There's just a small number of gaps.

This is something that we can now build a model with. How can we build a model? What we want to do is we want to create something which can predict for user 293, will they like movie 49, for example. So we've got to come up with some function that can represent that decision.

Here's a simple possible approach. We're going to take this idea of doing some matrix multiplications. So I've created here a random matrix. So here's one matrix of random numbers (the left). And I've created here another matrix of random numbers (the top). More specifically, for each movie, I've created five random numbers, and for each user, I've created five random numbers.

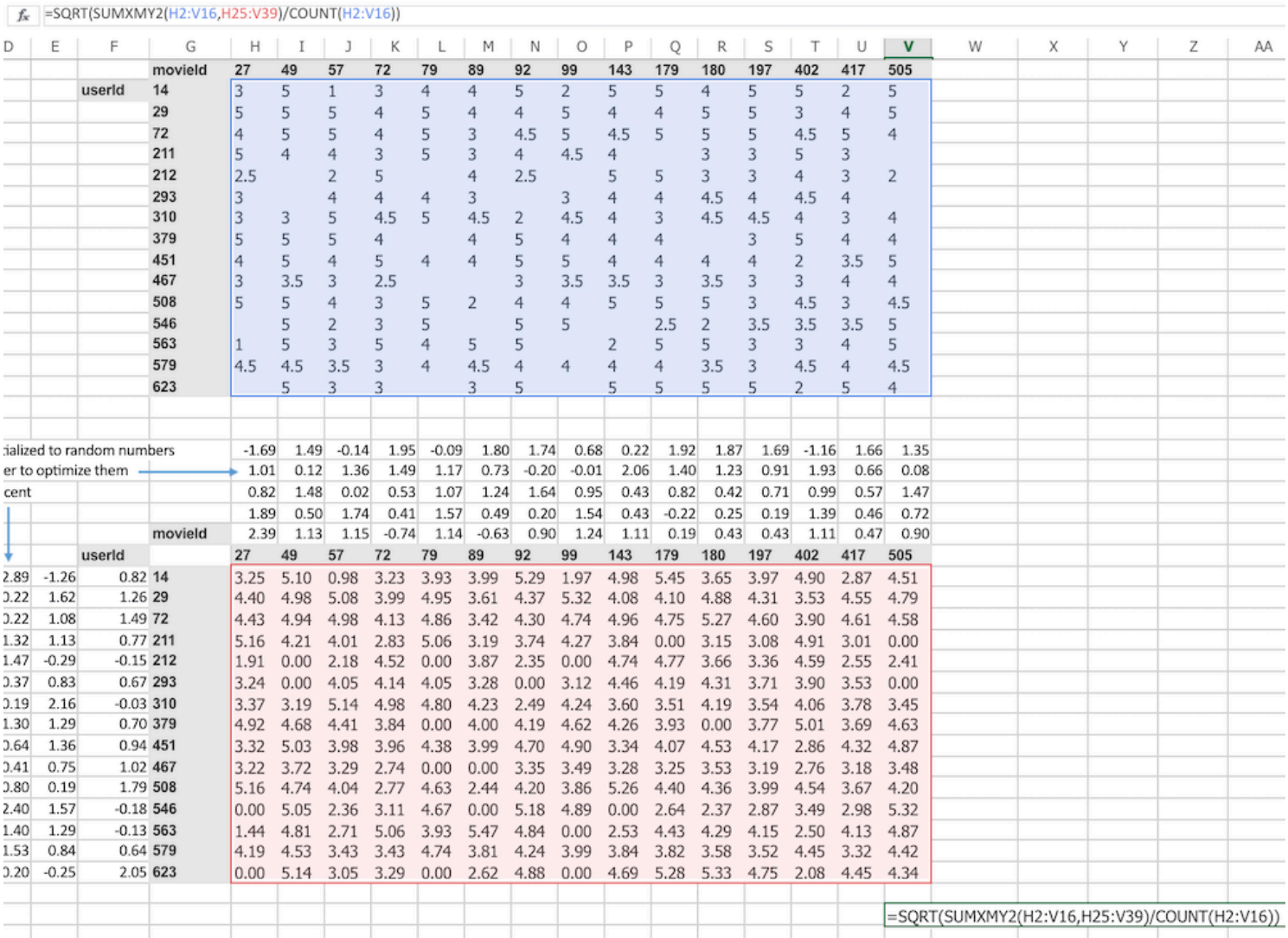
NB: These are initialized to random numbers
Then we use Solver to optimize them
with gradient descent

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | -1.69 | 1.49 | -0.14 | 1.95 | -0.09 | 1.80 | 1.74 | 0.68 | 0.22 | 1.92 |
| | | | | | | 1.01 | 0.12 | 1.36 | 1.49 | 1.17 | 0.73 | -0.20 | -0.01 | 2.06 | 1.40 |
| | | | | | | 0.82 | 1.48 | 0.02 | 0.53 | 1.07 | 1.24 | 1.64 | 0.95 | 0.43 | 0.82 |
| | | | | | | 1.89 | 0.50 | 1.74 | 0.41 | 1.57 | 0.49 | 0.20 | 1.54 | 0.43 | -0.22 |
| | | | | | movieId | 2.39 | 1.13 | 1.15 | -0.74 | 1.14 | -0.63 | 0.90 | 1.24 | 1.11 | 0.19 |
| | | | | userId | | 27 | 49 | 57 | 72 | 79 | 89 | 92 | 99 | 143 | 179 |
| 0.21 | 1.61 | 2.89 | -1.26 | 0.82 | 14 | =IF(H2="",0,MMULT($B25:$F25,H$19:H$23)) | | | | | | | | 4.98 | 5.45 |
| 1.55 | 0.75 | 0.22 | 1.62 | 1.26 | 29 | 4.40 | 4.98 | 5.08 | 3.99 | 4.95 | 3.61 | 4.37 | 5.32 | 4.08 | 4.10 |
| 1.50 | 1.17 | 0.22 | 1.08 | 1.49 | 72 | 4.43 | 4.94 | 4.98 | 4.13 | 4.86 | 3.42 | 4.30 | 4.74 | 4.96 | 4.75 |
| 0.47 | 0.89 | 1.32 | 1.13 | 0.77 | 211 | 5.16 | 4.21 | 4.01 | 2.83 | 5.06 | 3.19 | 3.74 | 4.27 | 3.84 | 0.00 |
| 0.31 | 2.10 | 1.47 | -0.29 | -0.15 | 212 | 1.91 | 0.00 | 2.18 | 4.52 | 0.00 | 3.87 | 2.35 | 0.00 | 4.74 | 4.77 |
| 1.00 | 1.45 | 0.37 | 0.83 | 0.67 | 293 | 3.24 | 0.00 | 4.05 | 4.14 | 4.05 | 3.28 | 0.00 | 3.12 | 4.46 | 4.19 |
| 1.16 | 1.16 | 0.19 | 2.16 | -0.03 | 310 | 3.37 | 3.19 | 5.14 | 4.98 | 4.80 | 4.23 | 2.49 | 4.24 | 3.60 | 3.51 |
| 0.79 | 1.07 | 1.30 | 1.29 | 0.70 | 379 | 4.92 | 4.68 | 4.41 | 3.84 | 0.00 | 4.00 | 4.19 | 4.62 | 4.26 | 3.93 |

So we could say, then, that user 14, movie 27; did they like it or not? Well, the rating, what we could do would be to multiply together this vector (red) and that vector (purple). We could do a dot product, and here's the dot product. Then we can basically do that for every possible thing in here. And thanks to spreadsheets, we can just do that in one place and copy it over, and it fills in the whole thing for us. Why would we do it this way? Well, this is

the basic starting point of a neural net, isn't it? A basic starting point of a neural net is that you take the matrix multiplication of two matrices, and that's what your first layer always is. So we just have to come up with some way of saying what are two matrices that we can multiply. Clearly, you need a vector for a user (a matrix for all the users) and a vector for a movie (a matrix for all the movies) and multiply them together, and you get some numbers. So they don't mean anything yet. They're just random. But we can now use gradient descent to try to make these numbers (top) and these numbers (left) give us results that are closer to what we wanted.

So how do we do that? Well, we set this up now as a linear model, so the next thing we need is a loss function. We can calculate our loss function by saying well okay movie 27 for user ID 14 should have been a rating of 3. With this random matrices, it's actually a rating of 0.91, so we can find the sum of squared errors would be $(3 - 0.91)^2$ and then we can add them up. So there's actually a sum squared in Excel already sum X minus y squared (`SUMXMY2`), so we can use just sum X minus y squared function, passing in those two ranges and then divide by the count to get the mean.




Here is a number that is the square root of the mean squared error. You sometimes you'll see people talk about MSE so that's the Mean Squared Error, sometimes you'll see RMSE that's the Root Mean Squared Error. Since I've got a square root at the front, this is the square root mean square error.

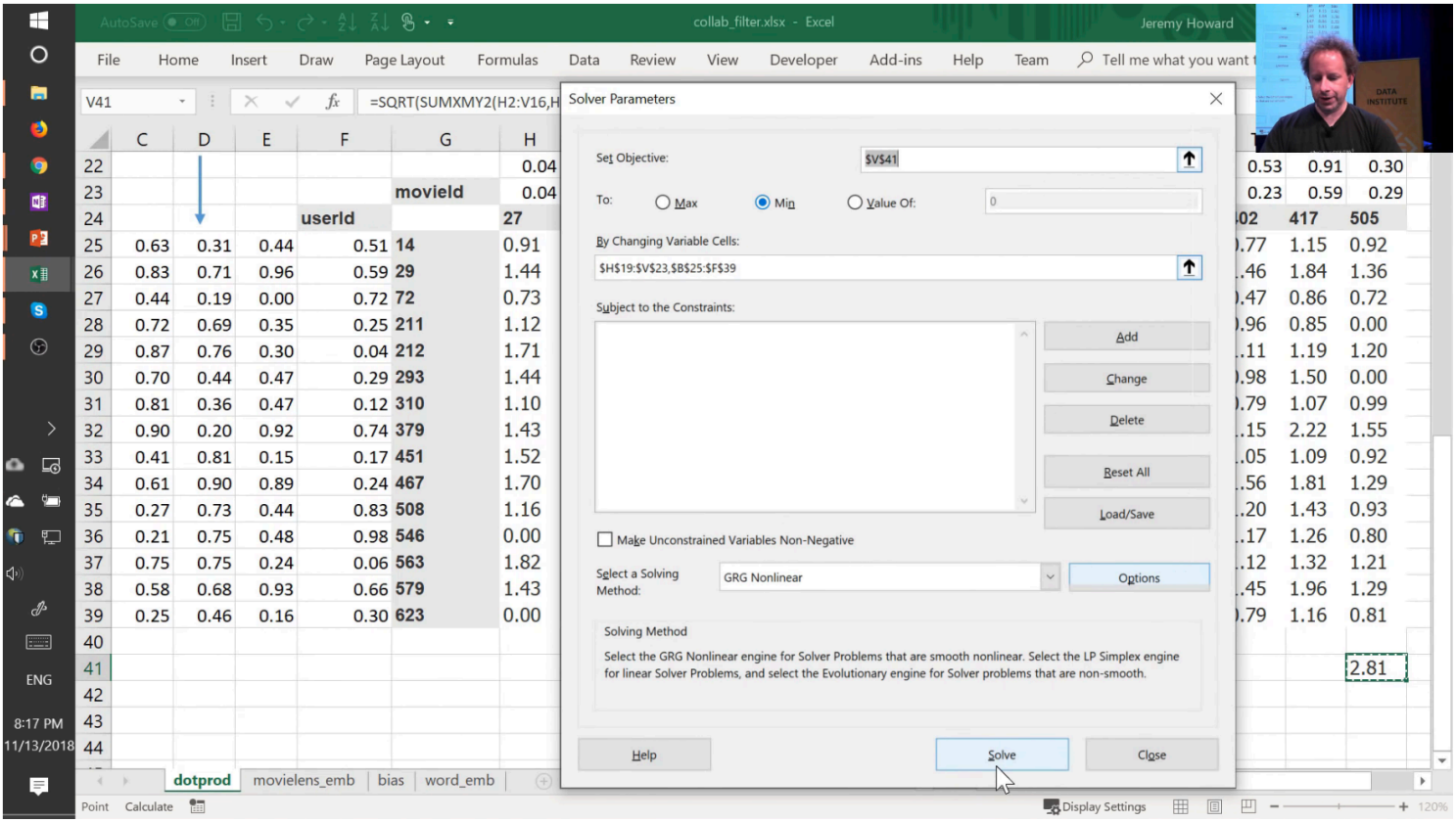
**Excel Solver [1:14:30]**

We have a loss, so now all we need to do is use gradient descent to try to modify our weight matrices to make that

loss smaller. Excel will do that for me.

If you don't have solver, go to Excel Options → Add-ins, and enable "Solver Add-in".

The gradient descent solver in Excel is called "Solver" and it just does normal gradient descent. You just go Data → Solver (you need to make sure that in your settings that you've enabled the solver extension which comes with Excel) and all you need to do is say which cell represents my loss function. So there it is, cell V41. Which cells contain your variables, and so you can see here, I've got H19 to V23 which is up here, and B25 to F39 which is over there, then you can just say "okay, set your loss function to a minimum by changing those cells" and click on Solve:




You'll see the starts a 2.81, and you can see the numbers going down. And all that's doing is using gradient descent exactly the same way that we did when we did it manually in the notebook the other day. But it's rather than solving the mean squared error for `a@x` in Python, instead it is solving the loss function here which is the mean squared error of the dot product of each of those vectors by each of these vectors.

We'll let that run for a little while and see what happens. But basically in micro, here is a simple way of creating a neural network which is really in this case, it's like just a single linear layer with gradient descent to solve a collaborative filtering problem.

Back to the notebook [1:17:02]

Let's go back and see what we do over here.

```
data = CollabDataBunch.from_df(ratings, seed=42)

y_range = [0,5.5]
```

```
learn = collab_learner(data, n_factors=50, y_range=y_range)

learn.fit_one_cycle(3, 5e-3)

Total time: 00:04
epoch  train_loss  valid_loss
1      1.600185    0.962681    (00:01)
2      0.851333    0.678732    (00:01)
3      0.660136    0.666290    (00:01)
```

So over here we used `collab_learner` to get a model. So the the function that was called in the notebook was `collab_learner` and as you dig deeper into deep learning, one of the really good ways to dig deeper into deep learning is to dig into the fastai source code and see what's going on. So if you're going to be able to do that, you need to know how to use your editor well enough to dig through the source code. Basically there are two main things you need to know how to do:

1. Jump to a particular "symbol", like a particular class or function by its name
2. When you're looking at a particular symbol, to be able to jump to its implementation

For example in this case, I want to find `def collab_learner`. In most editors including the one I use, vim, you can set it up so that you can hit tab or something and it jumps through all the possible completions, and you can hit enter and it jumps straight to the definition for you. So here is the definition of `collab_learner`. As you can see, it's pretty small as these things tend to be, and the key thing it does is to create model of a particular kind which is an `EmbeddingDotBias` model passing in the various things you asked for. So you want to find out in your editor how you jump to the definition of that, which in vim you just hit `Ctrl+]` and here is the definition of `EmbeddingDotBias`.

```
class EmbeddingDotBias(nn.Module):
    "Base dot model for collaborative filtering."
    def __init__(self, n_factors:int, n_users:int, n_items:int, y_range:Tuple[float,float]=None):
        super().__init__()
        self.y_range = y_range
        (self.u_weight, self.i_weight, self.u_bias, self.i_bias) = [embedding(*o) for o in [
            (n_users, n_factors), (n_items, n_factors), (n_users,1), (n_items,1)
        ]]

    def forward(self, users:LongTensor, items:LongTensor) -> Tensor:
        dot = self.u_weight(users)* self.i_weight(items)
        res = dot.sum(1) + self.u_bias(users).squeeze() + self.i_bias(items).squeeze()
        if self.y_range is None: return res
        return torch.sigmoid(res) * (self.y_range[1]-self.y_range[0]) + self.y_range[0]

class CollabDataBunch(DataBunch):…

class CollabLearner(Learner):…

def collab_learner(data, n_factors:int=None, use_nn:bool=False, metrics=None,
                   emb_szs:Dict[str,int]=None, wd:float=0.01, **kwargs)->Learner:
    "Create a Learner for collaborative filtering on `data`."
    emb_szs = data.get_emb_szs(ifnone(emb_szs, {}))
    u,m = data.classes.values()
    if use_nn: model = EmbeddingNN(emb_szs=emb_szs, **kwargs)
    else:      model = EmbeddingDotBias(n_factors, len(u), len(m), **kwargs)
    return CollabLearner(data, model, metrics=metrics, wd=wd)
```

Now we have everything on screen at once, and as you can see there's not much going on. The models that are being created for you by fastai are actually PyTorch models. And a PyTorch model is called an `nn.Module` that's the name in PyTorch of their models. It's a little more nuanced than that, but that's a good starting point for now. When a PyTorch `nn.Module` is run (when you calculate the result of that layer, neural net, etc), specifically, it always calls a method for you called `forward`. So it's in here that you get to find out how this thing is actually calculated.

When the model is built at the start, it calls this thing called `__init__` as we've briefly mentioned before in Python people tend to call this "dunder init". So dunder init is how we create the model, and forward is how we run the model.

One thing if you're watching carefully, you might notice is there's nothing here saying how to calculate the gradients of the model, and that's because PyTorch does it for us. So you only have to tell it how to calculate the output of your model, and PyTorch will go ahead and calculate the gradients for you.

So in this case, the model contains:

- a set of weights for a user
- a set of weights for an item
- a set of biases for a user
- a set of biases for an item

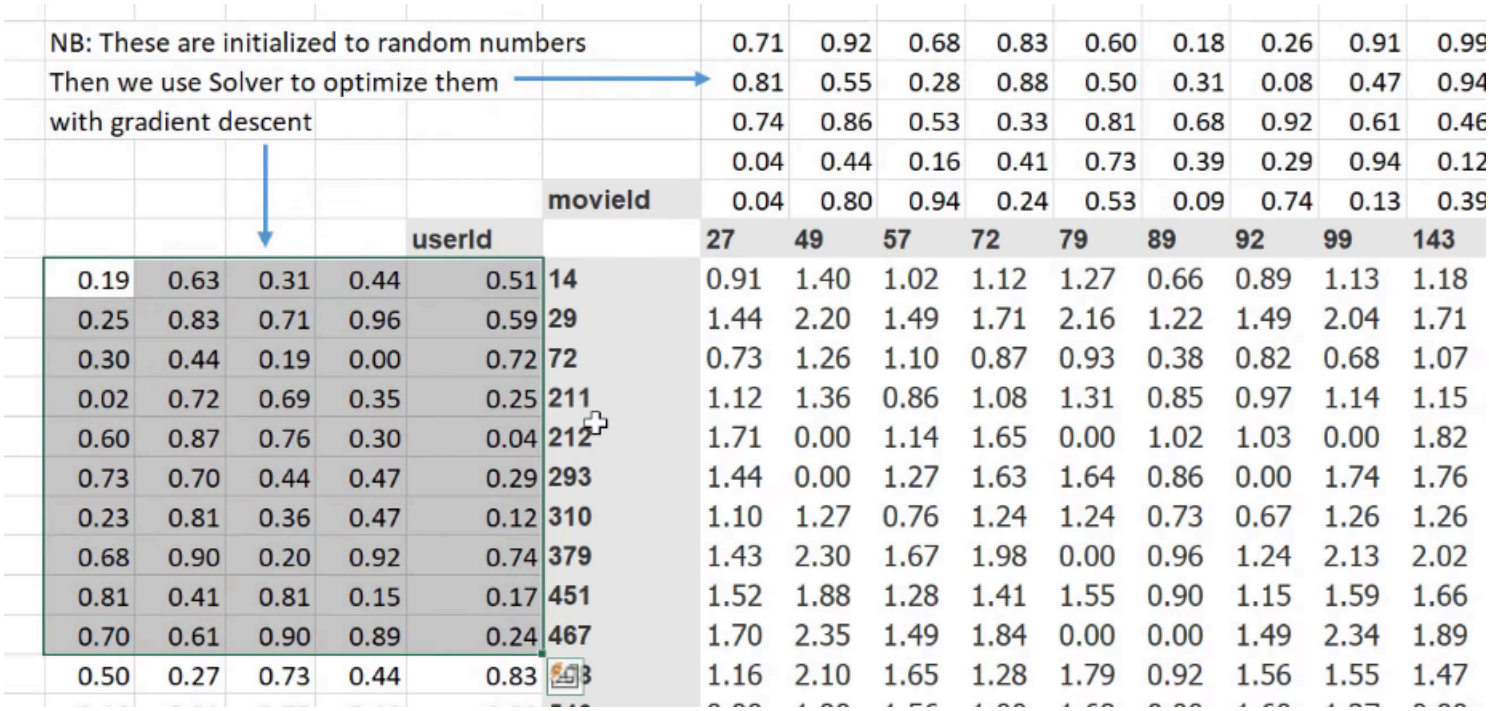
And each one of those is coming from this thing called `embedding`. Here is the definition of `embedding`:

```python
def embedding(ni:int,nf:int) -> nn.Module:
    "Create an embedding layer."
    emb = nn.Embedding(ni, nf)
    # See https://arxiv.org/abs/1711.09160
    with torch.no_grad(): trunc_normal_(emb.weight, std=0.01)
    return emb
```

All it does is it calls this PyTorch thing called `nn.Embedding`. In PyTorch, they have a lot of standard neural network layers set up for you. So it creates an embedding. And then this thing here (`trunc_normal_`) is it just randomizes it. This is something which creates normal random numbers for the embedding.

**Embedding [1:21:41]**

So what's an embedding? An embedding, not surprisingly, is a matrix of weights. Specifically, an embedding is a matrix of weights that looks something like this:




| | | | | | | 0.71 | 0.92 | 0.68 | 0.83 | 0.60 | 0.18 | 0.26 | 0.91 | 0.99 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | 0.81 | 0.55 | 0.28 | 0.88 | 0.50 | 0.31 | 0.08 | 0.47 | 0.94 |
| | | | | | | 0.74 | 0.86 | 0.53 | 0.33 | 0.81 | 0.68 | 0.92 | 0.61 | 0.46 |
| | | | | | | 0.04 | 0.44 | 0.16 | 0.41 | 0.73 | 0.39 | 0.29 | 0.94 | 0.12 |
| | | | | | movieId | 0.04 | 0.80 | 0.94 | 0.24 | 0.53 | 0.09 | 0.74 | 0.13 | 0.39 |
| | | | | userId | | 27 | 49 | 57 | 72 | 79 | 89 | 92 | 99 | 143 |
| 0.19 | 0.63 | 0.31 | 0.44 | 0.51 | 14 | 0.91 | 1.40 | 1.02 | 1.12 | 1.27 | 0.66 | 0.89 | 1.13 | 1.18 |
| 0.25 | 0.83 | 0.71 | 0.96 | 0.59 | 29 | 1.44 | 2.20 | 1.49 | 1.71 | 2.16 | 1.22 | 1.49 | 2.04 | 1.71 |
| 0.30 | 0.44 | 0.19 | 0.00 | 0.72 | 72 | 0.73 | 1.26 | 1.10 | 0.87 | 0.93 | 0.38 | 0.82 | 0.68 | 1.07 |
| 0.02 | 0.72 | 0.69 | 0.35 | 0.25 | 211 | 1.12 | 1.36 | 0.86 | 1.08 | 1.31 | 0.85 | 0.97 | 1.14 | 1.15 |
| 0.60 | 0.87 | 0.76 | 0.30 | 0.04 | 212 | 1.71 | 0.00 | 1.14 | 1.65 | 0.00 | 1.02 | 1.03 | 0.00 | 1.82 |
| 0.73 | 0.70 | 0.44 | 0.47 | 0.29 | 293 | 1.44 | 0.00 | 1.27 | 1.63 | 1.64 | 0.86 | 0.00 | 1.74 | 1.76 |
| 0.23 | 0.81 | 0.36 | 0.47 | 0.12 | 310 | 1.10 | 1.27 | 0.76 | 1.24 | 1.24 | 0.73 | 0.67 | 1.26 | 1.26 |
| 0.68 | 0.90 | 0.20 | 0.92 | 0.74 | 379 | 1.43 | 2.30 | 1.67 | 1.98 | 0.00 | 0.96 | 1.24 | 2.13 | 2.02 |
| 0.81 | 0.41 | 0.81 | 0.15 | 0.17 | 451 | 1.52 | 1.88 | 1.28 | 1.41 | 1.55 | 0.90 | 1.15 | 1.59 | 1.66 |
| 0.70 | 0.61 | 0.90 | 0.89 | 0.24 | 467 | 1.70 | 2.35 | 1.49 | 1.84 | 0.00 | 0.00 | 1.49 | 2.34 | 1.89 |
| 0.50 | 0.27 | 0.73 | 0.44 | 0.83 | 3 | 1.16 | 2.10 | 1.65 | 1.28 | 1.79 | 0.92 | 1.56 | 1.55 | 1.47 |

It's a matrix of weights which you can basically look up into, and grab one item out of it. So basically an embedding matrix is just a weight matrix that is designed to be something that you index into it as an array, and grab one vector out of it. That's what an embedding matrix is. In our case, we have an embedding matrix for a user and an embedding matrix for a movie. And here, we have been taking the dot product of them:

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NB: These are initialized to random numbers | | | | | | 0.71 | 0.92 | 0.68 | 0.83 | 0.60 | 0.18 | 0.26 | 0.91 | 0.99 | 0.52 |
| Then we use Solver to optimize them → | | | | | | 0.81 | 0.55 | 0.28 | 0.88 | 0.50 | 0.31 | 0.08 | 0.47 | 0.94 | 0.70 |
| with gradient descent | | | | | | 0.74 | 0.86 | 0.53 | 0.33 | 0.81 | 0.68 | 0.92 | 0.61 | 0.46 | 0.64 |
| | | | | | | 0.04 | 0.44 | 0.16 | 0.41 | 0.73 | 0.39 | 0.29 | 0.94 | 0.12 | 0.67 |
| | | | | | movieId | 0.04 | 0.80 | 0.94 | 0.24 | 0.53 | 0.09 | 0.74 | 0.13 | 0.39 | 0.44 |
| | | | | userId | | 27 | 49 | 57 | 72 | 79 | 89 | 92 | 99 | 143 | 179 |
| 0.19 | 0.63 | 0.31 | 0.44 | 0.51 | 14 | 0.91 | 1.40 | 1.02 | 1.12 | 1.27 | 0.66 | 0.89 | 1.13 | 1.18 | 1.27 |
| 0.25 | 0.83 | 0.71 | 0.96 | 0.59 | 29 | 1.44 | 2.20 | 1.49 | 1.71 | 2.16 | 1.22 | 1.49 | 2.04 | 1.71 | 2.08 |
| 0.30 | 0.44 | 0.19 | 0.00 | 0.72 | 72 | 0.73 | 1.26 | 1.10 | 0.87 | 0.93 | 0.38 | 0.82 | 0.68 | 1.07 | 0.90 |
| 0.02 | 0.72 | 0.69 | 0.35 | 0.25 | 211 | 1.12 | 1.36 | 0.86 | 1.08 | 1.31 | 0.85 | 0.97 | 1.14 | 1.15 | 0.00 |
| 0.60 | 0.87 | 0.76 | 0.30 | 0.04 | 212 | 1.71 | 0.00 | 1.14 | 1.65 | 0.00 | 1.02 | 1.03 | 0.00 | 1.82 | 1.64 |
| 0.73 | 0.70 | 0.44 | 0.47 | 0.29 | 293 | 1.44 | 0.00 | 1.27 | =IF(K7="",0,MMULT($B30:$F30,K$19:K$23)) | | | | | | |
| 0.23 | 0.81 | 0.36 | 0.47 | 0.12 | 310 | 1.10 | 1.27 | 0.76 | 1.24 | 1.24 | 0.73 | 0.67 | 1.26 | 1.26 | 1.29 |
| 0.68 | 0.90 | 0.20 | 0.92 | 0.74 | 379 | 1.43 | 2.30 | 1.67 | 1.98 | 0.00 | 0.96 | 1.24 | 2.13 | 2.02 | 2.07 |

But if you think about it, that's not quite enough. Because we're missing this idea that maybe there are certain movies that everybody likes more. Maybe there are some users that just tend to like movies more. So I don't really just want to multiply these two vectors together, but I really want to add a single number of like how popular is this movie, and add a single number of like how much does this user like movies in general. So those are called "bias" terms. Remember how I said there's this idea of bias and the way we dealt with that in our gradient descent notebook was we added a column of 1's. But what we tend to do in practice is we actually explicitly say I want to add a bias term. So we don't just want to have prediction equals dot product of these two things, we want to say it's the dot product of those two things plus a bias term for a movie plus a bias term for user ID.

**Back to code [1:23:55]**

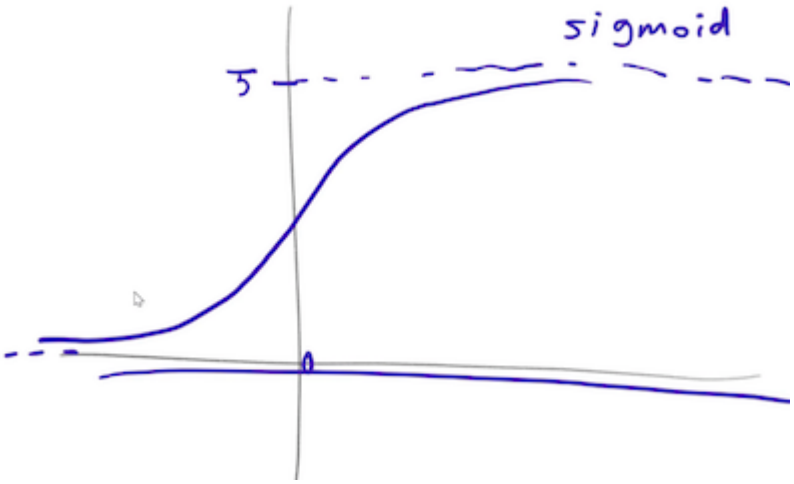
So that's basically what happens. We when we set up the model, we set up the embedding matrix for the users and the embedding matrix for the items. And then we also set up the bias vector for the users and the bias vector for the items.

```
class EmbeddingDotBias(nn.Module):
    "Base dot model for collaborative filtering."
    def __init__(self, n_factors:int, n_users:int, n_items:int, y_range:Tuple[float,float]=None):
        super().__init__()
        self.y_range = y_range
        (self.u_weight, self.i_weight, self.u_bias, self.i_bias) = [embedding(*o) for o in [
            (n_users, n_factors), (n_items, n_factors), (n_users,1), (n_items,1)
        ]]

    def forward(self, users:LongTensor, items:LongTensor) -> Tensor:
        dot = self.u_weight(users)* self.i_weight(items)
        res = dot.sum(1) + self.u_bias(users).squeeze() + self.i_bias(items).squeeze()
        if self.y_range is None: return res
        return torch.sigmoid(res) * (self.y_range[1]-self.y_range[0]) + self.y_range[0]
```

Then when we calculate the model, we literally just multiply the two together. Just like we did. We just take that product, we call it `dot`. Then we add the bias, and (putting aside `y_range` for a moment) that's what we return. So you can see that our model is literally doing what we did in the spreadsheet with the tweak that we're also adding the bias. So it's an incredibly simple linear model. For these kinds of collaborative filtering problems, this kind of simple linear model actually tends to work pretty well.

Then there's one tweak that we do at the end which is that in our case we said that there's y range of between 0 and 5.5. So here's something to point out. So you do that dot product and you add on the two biases and that could give you any possible number along the number line from very negative through to very positive numbers. But we know that we always want to end up with a number between zero and five. What if we mapped that number line like so, to this function. The shape of that function is called a sigmoid. And so, it's gonna asymptote to five and it's gonna asymptote to zero.




That way, whatever number comes out of our dot product and adding the biases, if we then stick it through this function, it's never going to be higher than 5 and never going to be smaller than 0. Now strictly speaking, that's not necessary. Because our parameters could learn a set of weights that gives about the right number. So why would we do this extra thing if it's not necessary? The reason is, we want to make its life as easy for our model as possible. If we actually set it up so it's impossible for it to ever predict too much or too little, then it can spend more of its weights predicting the thing we care about which is deciding who's going to like what movie. So this is an idea we're going to keep coming back to when it comes to like making neural network's work better. It's about all these little decisions that we make to basically make it easier for the network to learn the right thing. So that's the last tweak here:

```
return torch.sigmoid(res) * (self.y_range[1]-self.y_range[0]) + self.y_range[0]
```

We take the result of this dot product plus biases, we put it through a sigmoid. A sigmoid is just a function which is basically $\frac{1}{1+e^{-x}}$, but the definition doesn't much matter. But it just has the shape that I just mentioned, and that goes between 0 and 1. If you then multiply that by `y_range[1]` minus `y_range[0]` plus `y_range[0]`, then that's going to give you something that's between `y_range[0]` and `y_range[1]`.

So that means that this tiny little neural network, I mean it's a push to call it a neural network. But it is a neural network with one weight matrix and no nonlinearities. So it's kind of the world's most boring neural network with a sigmoid at the end. I guess it does have a non-linearity. The sigmoid at the end is the non-linearity, it only has one layer of weights. That actually turns out to give close to state-of-the-art performance. I've looked up online to find out like what are the best results people have on this MovieLens 100k database, and the results I get from this little thing is better than any of the results I can find from the standard commercial products that you can download that are specialized for this. And the trick seems to be that adding this little sigmoid makes a big difference.

[1:29:09]

**Question**: There was a question about how you set up your vim, and I've already linked to your .vimrc but I wanted know if you had more to say about. They really like your setup ?

You like my setup? There's almost nothing in my setup. It's pretty bare honestly. I mean whatever you're doing with your editor, you probably want it to look like this which is when you've got a class that you're not currently working on it should be this is called folded/folding?it should be closed up so you can't see it. So you basically want something where it's easy to close and open folds, so vim already does all this for you. Then as I mentioned, you also want something where you can jump to the definition of things which in vim called using tags (e.g. to jump to the definition of `Learner`, position the cursor over `Learner` and hit `Ctrl+]`). Basically vim already does all this for you. You just have to read instructions. My .vimrc is minimal. I basically hardly use any extensions or anything. Another great editor to use is a Visual Studio Code. It's free and it's awesome and it has all the same features that you're seeing that vim does, basically VS Sode does all of those things as well. I quite like using vim because I can use it on the remote machine and play around, but you can of course just clone the git repo into your local computer and open it up with VS Code to play around with. Just don't try and look through the code just on github or something. That's going to drive you crazy. You need to be able to open it and close it and jump and jump back. Maybe people can create some threads on the forum for vim tips, VS Code tips, Sublime tips, whatever. For me, I would if you're gonna pick an editor, if you want to use something on your local, I would go with the VS Code today. I think it's the best. If you want to use something on the terminal side, I would go with VIM or Emacs, to me they're clear winners.

## Overview of important terminology [1:31:24]

So what I wanted to close with today is, to take this collaborative filtering example and describe how we're going to build on top of it for the next three lessons to create the more complex neural networks we've been seeing. Roughly speaking, this is the bunch of concepts that we need to learn about:

- Inputs
- Weights/parameters
    - Random
- Activations
- Activation functions / nonlinearities
- Output
- Loss
- Metric
- Cross-entropy
- Softmax
- Fine tuning
    - Layer deletion and random weights
    - Freezing & unfreezing

Let's think about what happens when you're using a neural network to do image recognition. Let's take a single pixel. You've got lots of pixels, but let's take a single pixel. So you've got a red a green and a blue pixel. Each one of those is some number between 0 and 255, or we normalize them so they have the mean of zero and standard deviation of one. But let's just do 0 to 255 version. So red: 10, green: 20, blue 30. So what do we do with these? Well, what we do is we basically treat that as a vector, and we multiply it by a matrix. So this matrix (depending on how you think of the rows and the columns), let's treat the matrix is having three rows and then how many columns? You get to pick. Just like with the collaborative filtering version, I decided to pick a vector of size five for each of my embedding vectors. So that would mean that's an embedding of size 5. You get to pick how big your weight matrix is. So let's make it size 5. This is 3 by 5.

Initially, this weight matrix contains random numbers. Remember we looked at `embedding` weight matrix just now?

```python
def embedding(ni:int,nf:int) -> nn.Module:
    "Create an embedding layer."
    emb = nn.Embedding(ni, nf)
    # See https://arxiv.org/abs/1711.09160
    with torch.no_grad(): trunc_normal_(emb.weight, std=0.01)
    return emb
```

There were two lines; the first line was create the matrix, and the second was fill it with random numbers? That's all we do. I mean it all gets hidden behind the scenes by fastai and PyTorch, but that's all it's doing. So it's creating a matrix of random numbers when you set it up. The number of rows has to be 3 to match the input, and the number of columns can be as big as you like. So after you multiply the input vector by that weight matrix, you're going to end up with a vector of size 5.

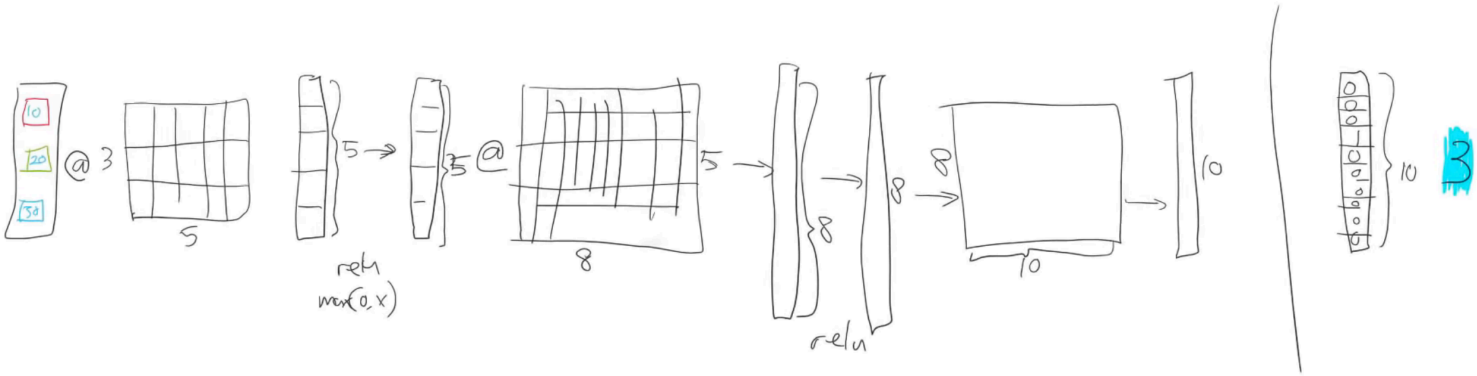
People often ask how much linear algebra do I need to know to be able to do deep learning. This is the amount you need. And if you're not familiar with this, that's fine. You need to know about matrix products. You don't need to know a lot about them, you just need to know like computationally what are they and what do they do. You've got to be very comfortable with if a matrix of size blah times a matrix of size blah gives a matrix or size blah (i.e. how do the dimensions match up). So if you have 3, and they remember in numpy and PyTorch, we use @ times 3 by 5 gives a vector of size 5.

Then what happens next; it goes through an activation function such as ReLU which is just `max(0,x)` and spits out a new vector which is, of course, going to be exactly the same size because **no activation function changes the size?it only changes the contents**. So that's still of size 5.

What happens next? We multiply by another matrix. Again, it can be any number of columns, but the number of rows has to map nicely. So it's going to be 5 by whatever. Maybe this one has 5, let's say, by 10. That's going to give some output?it should be size 10 and again we put that through ReLU, and again that gives us something of the same size.

Then we can put that through another matrix. Actually, just to make this a bit clearer (you'll see why in a moment), I'm going to use 8 not 10.

Let's say we're doing digit recognition. There are ten possible digits, so my last weight matrix has to be 10 in size. Because then that's going to mean my final output is a vector of 10 in size. Remember if you're doing that digit recognition, we take our actuals which is 10 in size. And if the number we're trying to predict was the number 3, then that means that there is a 1 in the third position ([0,0,0,1,0,…]).



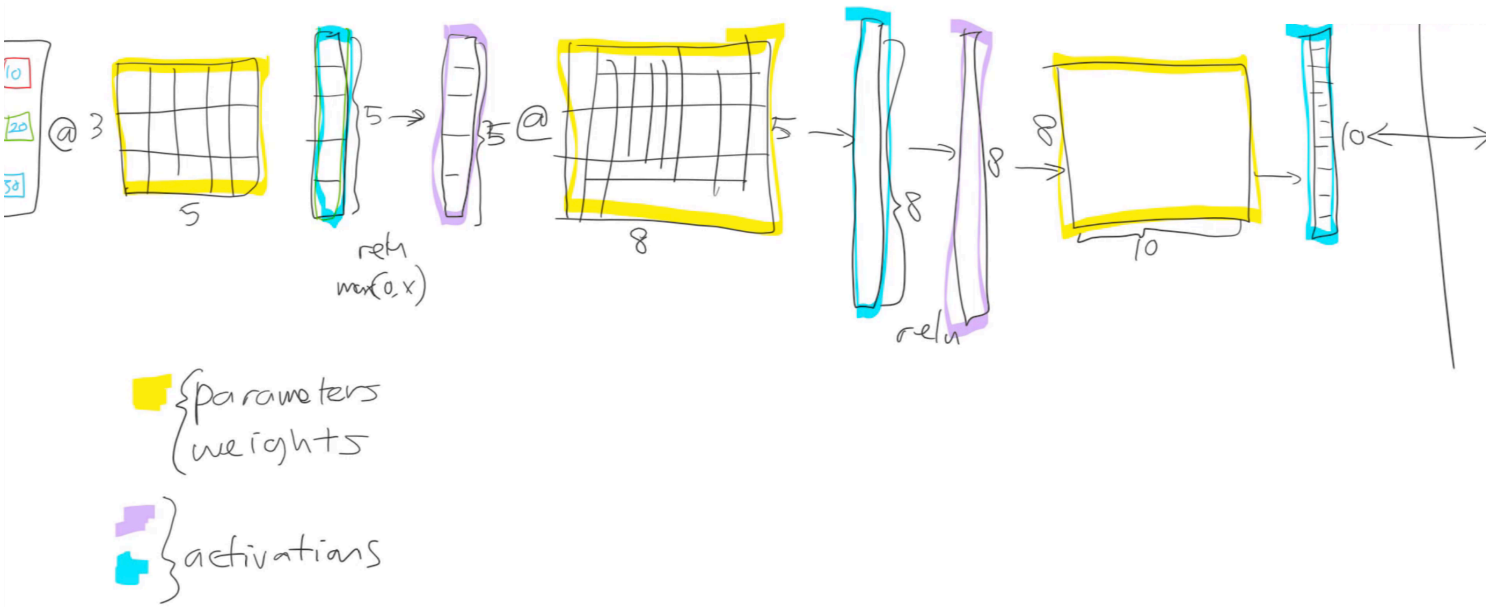

So what happens is our neural net runs along starting with our input, and going weight matrix→ReLU→ weight matrix→ReLU→ weight matrix→ final output. Then we compare these two together to see how close they are (i.e. how close they match) using some loss function and we'll learn about all the loss functions that we use next week. For now, the only one we've learned is mean squared error. And we compare the output (you can think of

them as probabilities for each of the 10) to the actual each of the 10 to get a loss, and then we find the gradients of every one of the weight matrices with respect to that, and we update the weight matrices.

The main thing I wanted to show right now is the terminology we use because it's really important.

These things (yellow) contain numbers. Specifically they initially are matrices containing random numbers. And we can refer to these yellow things, in PyTouch, they're called parameters. Sometimes we'll refer to them as weights, although weights is slightly less accurate because they can also be biases. But we kind of use the terms a little bit interchangeably. Strictly speaking, we should call them parameters.

Then after each of those matrix products, that calculates a vector of numbers. Here are some numbers (blue) that are calculated by a weight matrix multiply. And then there's some other set of numbers (purple) that are calculated as a result of a ReLU as well as the activation function. Either one is called activations.

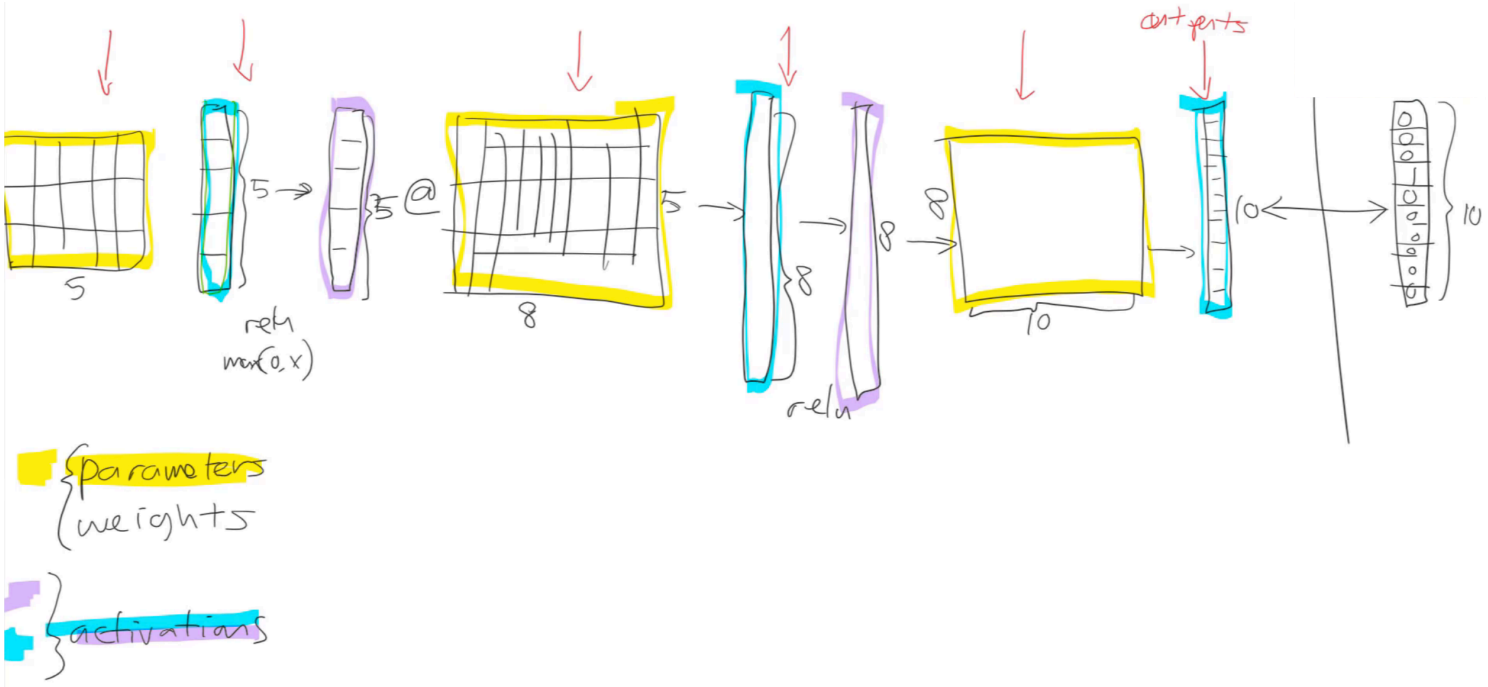



Activations and parameters, both refer to numbers. They are numbers. But **Parameters** are numbers that are stored, they are used to make a calculation. **Activations** are the result of a calculation?the numbers that are calculated. So they're the two key things you need to remember.

So use these terms, and use them correctly and accurately. And if you read these terms, they mean these very specific things. So don't mix them up in your head. And remember, they're nothing weird and magical?they are very simple things.

- An activation is the result of either a matrix multiply or an activation function.
- Parameters are the numbers inside the matrices that we multiply by.

That's it. Then there are some special layers. Every one of these things that does a calculation, all of these things that does a calculation (red arrow), are all called layers. They're the layers of our neural net. So every layer results in a set of activations because there's a calculation that results in a set of results.




There's a special layer at the start which is called the input layer, and then at the end you just have a set of activations and we can refer to those special numbers (I mean they're not special mathematically but they're semantically special); we can call those the outputs. The important point to realize here is the outputs of a neural net are not actually mathematically special, they're just the activations of a layer.

So what we did in our collaborative filtering example, we did something interesting. We actually added an additional activation function right at the very end. We added an extra activation function which was sigmoid, specifically it was a scaled sigmoid which goes between 0 and 5. It's very common to have an activation function as your last layer, and it's almost never going to be a ReLU because it's very unlikely that what you actually want is something that truncates at zero. It's very often going to be a sigmoid or something similar because it's very likely that actually what you want is something that's between two values and kind of scaled in that way.

So that's nearly it. Inputs, weights, activations, activation functions (which we sometimes call nonlinearities), output, and then the function that compares those two things together is called the loss function, which so far we've used MSE.

That's enough for today. So what we're going to do next week is we're going to kind of add in a few more extra bits which is we're going to learn the loss function that's used for classification called **cross-entropy**, we're going to use the activation function that's used for single label classification called **softmax**, and we're also going to learn exactly what happens when we do fine-tuning in terms of how these layers actually, what happens with unfreeze, and what happens when we create transfer learning. Thanks everybody! Looking forward to seeing you next week.