```
pip install fastai
```

```
pip install fastbook
```

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

```
from google.colab import data_table
```

DataSet

Recommendation Systems

- Learning form past behavior

great dataset that we can use, called MovieLens. This dataset contains tens of millions of movie rankings (a combination of a movie ID, a user ID, and a numeric rating), although we will just use a subset of 100,000 of them for our example.

```
path=untar_data(URLs.ML_100k)
```

```
100.15% [4931584/4924029 00:00<00:00]
```

```
Path.BASE_PATH=path
```

```
path.ls()
```

```
(#23)
[Path('ua.base'),Path('u3.test'),Path('ub.test'),Path('ub.base'),Path('u4.base'),Path('ua.test'),
```

According to the README, the main table is in the file u.data. It is tab-separated and the columns are, respectively user, movie, rating, and timestamp.

```
ratings=pd.read_csv(
                    path/'u.data',delimiter='\t',header=None,
                    names=['user','movie','rating','timestamp'])

data_table.DataTable(ratings, include_index=False, num_rows_per_page=10,min_width='10px ')
```

```
Warning: total number of rows (100000) exceeds max_rows (20000). Limiting to first (20000) rows.
```

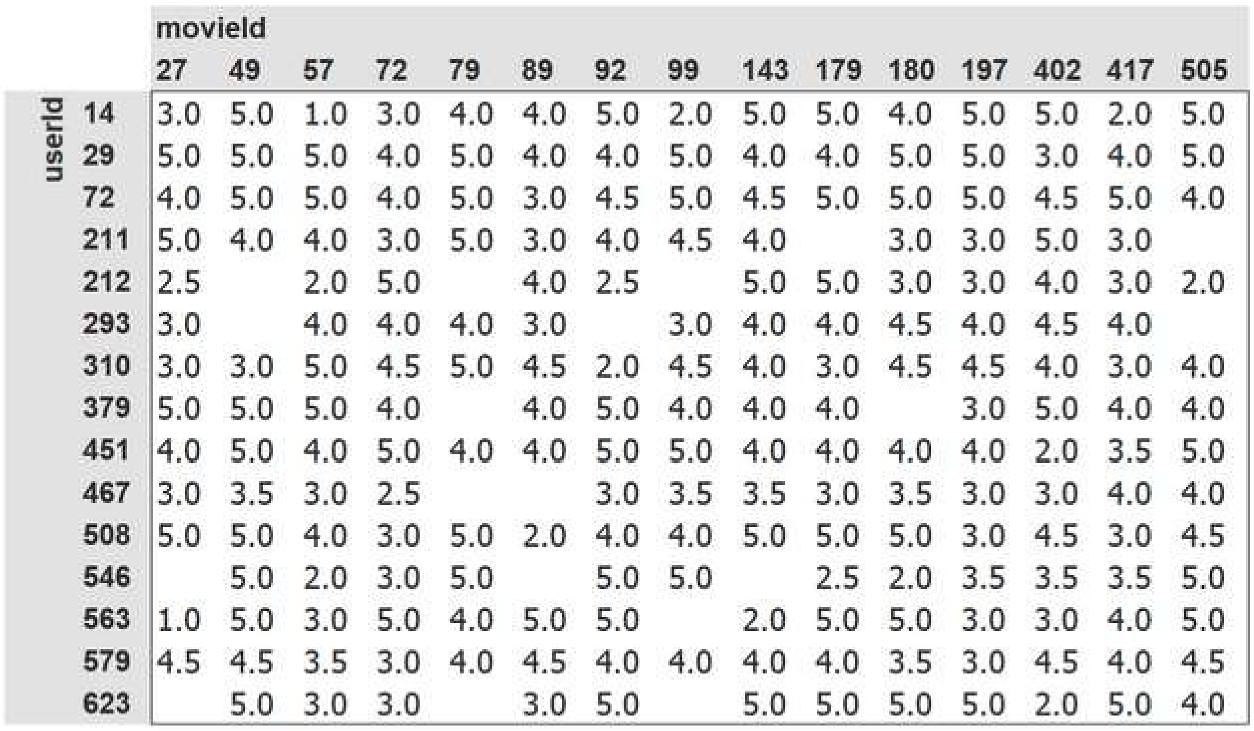
1 to 10 of 20000 entries Filter

user	movie	rating	timestamp
196	242	3	881250949
186	302	3	891717742
22	377	1	878887116
244	51	2	880606923
166	346	1	886397596
298	474	4	884182806
115	265	2	881171488
253	465	5	891628467
305	451	3	886324817
6	86	3	883603013

Show 10 per page 1 2 10 100 1000 1900 1990 2000

Fill these missing places

- recommend



userId	movieId 27	49	57	72	79	89	92	99	143	179	180	197	402	417	505
14	3.0	5.0	1.0	3.0	4.0	4.0	5.0	2.0	5.0	5.0	4.0	5.0	5.0	2.0	5.0
29	5.0	5.0	5.0	4.0	5.0	4.0	4.0	5.0	4.0	4.0	5.0	5.0	3.0	4.0	5.0
72	4.0	5.0	5.0	4.0	5.0	3.0	4.5	5.0	4.5	5.0	5.0	5.0	4.5	5.0	4.0
211	5.0	4.0	4.0	3.0	5.0	3.0	4.0	4.5	4.0		3.0	3.0	5.0	3.0	
212	2.5		2.0	5.0		4.0	2.5		5.0	5.0	3.0	3.0	4.0	3.0	2.0
293	3.0		4.0	4.0	4.0	3.0		3.0	4.0	4.0	4.5	4.0	4.5	4.0	
310	3.0	3.0	5.0	4.5	5.0	4.5	2.0	4.5	4.0	3.0	4.5	4.5	4.0	3.0	4.0
379	5.0	5.0	5.0	4.0		4.0	5.0	4.0	4.0	4.0		3.0	5.0	4.0	4.0
451	4.0	5.0	4.0	5.0	4.0	4.0	5.0	5.0	4.0	4.0	4.0	4.0	2.0	3.5	5.0
467	3.0	3.5	3.0	2.5			3.0	3.5	3.5	3.0	3.5	3.0	3.0	4.0	4.0
508	5.0	5.0	4.0	3.0	5.0	2.0	4.0	4.0	5.0	5.0	5.0	3.0	4.5	3.0	4.5
546		5.0	2.0	3.0	5.0		5.0	5.0		2.5	2.0	3.5	3.5	3.5	5.0
563	1.0	5.0	3.0	5.0	4.0	5.0	5.0		2.0	5.0	5.0	3.0	3.0	4.0	5.0
579	4.5	4.5	3.5	3.0	4.0	4.5	4.0	4.0	4.0	4.0	3.5	3.0	4.5	4.0	4.5
623		5.0	3.0	3.0		3.0	5.0		5.0	5.0	5.0	5.0	2.0	5.0	4.0

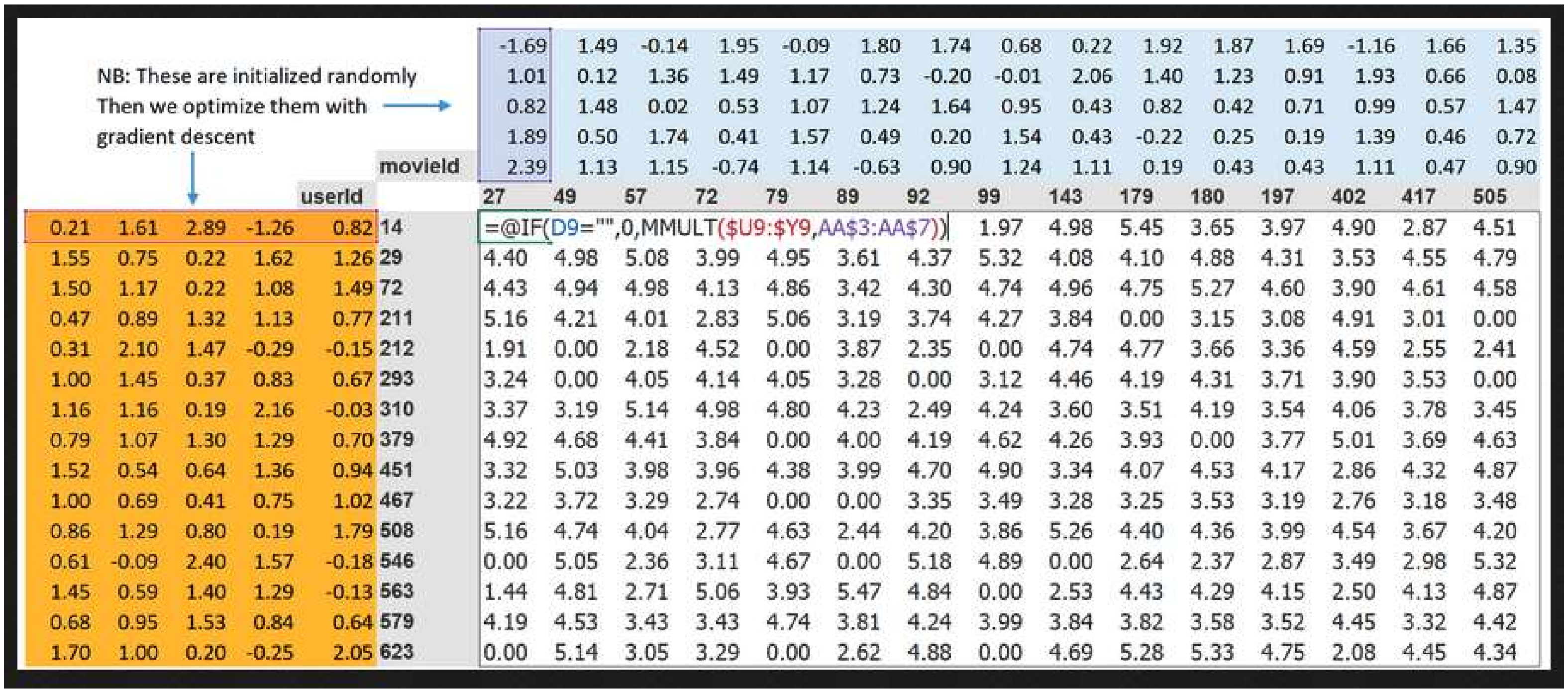
Problem

- we d not whta latent factors are
- we don not know hoe to score them for each user and movie
- **so we will learn them**

Learning the Latent Factors

There is surprisingly little difference between specifying the structure of a model, as we did in the last section, and learning one, since we can just use our general gradient descent approach.

Step 1 of this approach is to randomly initialize some parameters. These parameters will be a set of latent factors for each user and movie. We will have to decide how many to use. We will discuss how to select this shortly, but for illustrative purposes let's use 5 for now. Because each user will have a set of these factors and each movie will have a set of these factors, we can show these randomly initialized values right next to the users and movies in our crosstab, and we can then fill in the dot products for each of these combinations in the middle. For example, <> shows what it looks like in Microsoft Excel, with the top-left cell formula displayed as an example.



NB: These are initialized randomly
Then we 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	1.87	1.69	-1.16	1.66	1.35
						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
						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
						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
					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
				userId		27	49	57	72	79	89	92	99	143	179	180	197	402	417	505
0.21	1.61	2.89	-1.26	0.82	14	=@IF(D9="",0,MMULT($U9:$Y9,AA$3:AA$7))							1.97	4.98	5.45	3.65	3.97	4.90	2.87	4.51
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
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
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
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	3.66	3.36	4.59	2.55	2.41
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	4.31	3.71	3.90	3.53	0.00
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	4.19	3.54	4.06	3.78	3.45
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	0.00	3.77	5.01	3.69	4.63
1.52	0.54	0.64	1.36	0.94	451	3.32	5.03	3.98	3.96	4.38	3.99	4.70	4.90	3.34	4.07	4.53	4.17	2.86	4.32	4.87
1.00	0.69	0.41	0.75	1.02	467	3.22	3.72	3.29	2.74	0.00	0.00	3.35	3.49	3.28	3.25	3.53	3.19	2.76	3.18	3.48
0.86	1.29	0.80	0.19	1.79	508	5.16	4.74	4.04	2.77	4.63	2.44	4.20	3.86	5.26	4.40	4.36	3.99	4.54	3.67	4.20
0.61	-0.09	2.40	1.57	-0.18	546	0.00	5.05	2.36	3.11	4.67	0.00	5.18	4.89	0.00	2.64	2.37	2.87	3.49	2.98	5.32
1.45	0.59	1.40	1.29	-0.13	563	1.44	4.81	2.71	5.06	3.93	5.47	4.84	0.00	2.53	4.43	4.29	4.15	2.50	4.13	4.87
0.68	0.95	1.53	0.84	0.64	579	4.19	4.53	3.43	3.43	4.74	3.81	4.24	3.99	3.84	3.82	3.58	3.52	4.45	3.32	4.42
1.70	1.00	0.20	-0.25	2.05	623	0.00	5.14	3.05	3.29	0.00	2.62	4.88	0.00	4.69	5.28	5.33	4.75	2.08	4.45	4.34

Step 2

- of this approach is to calculate our predictions. As we've discussed, we can do this by simply taking the dot product of each movie with each user. If, for instance, the first latent user factor represents how much the user likes action movies and the first latent movie factor represents if the movie has a lot of action or not, the product of those will be particularly high if either the user likes action movies and the movie has a lot of action in it or the user doesn't like action movies and the movie doesn't have any action in it. On the other hand, if we have a mismatch (a user loves action movies but the movie isn't an action film, or the user doesn't like action movies and it is one), the product will be very low.

Step 3

- is to calculate our loss. We can use any loss function that we wish; let's pick mean squared error for now, since that is one reasonable way to represent the accuracy of a prediction.

That's all we need. With this in place, we can optimize our parameters (that is, the latent factors) using stochastic gradient descent, such as to minimize the loss. At each step, the stochastic gradient descent optimizer will calculate the match between each movie and each user using the dot product, and will compare it to the actual rating that each user gave to each movie. It will then calculate the derivative of this value and will step the weights by multiplying this by the learning rate. After doing this lots of times, the loss will get better and better, and the recommendations will also get better and better.

To use the usual Learner.fit function we will need to get our data into a DataLoaders, so let's focus on that now.

Creating The DatLoaders

When showing the data, we would rather see movie titles than their IDs. The table u.item contains the correspondence of IDs to titles:

```
movies = pd.read_csv(path/'u.item',  delimiter='|', encoding='latin-1',
                     usecols=(0,1), names=('movie','title'), header=None)
movies.head()
```

	movie	title
0	1	Toy Story (1995)
1	2	GoldenEye (1995)
2	3	Four Rooms (1995)
3	4	Get Shorty (1995)
4	5	Copycat (1995)

erge this with our ratings table to get the user ratings by title:

```
ratings = ratings.merge(movies)
ratings.head()
```

	user	movie	rating	timestamp	title
0	196	242	3	881250949	Kolya (1996)
1	63	242	3	875747190	Kolya (1996)
2	226	242	5	883888671	Kolya (1996)
3	154	242	3	879138235	Kolya (1996)
4	306	242	5	876503793	Kolya (1996)

DataLoader

```
dls = CollabDataLoaders.from_df(ratings, item_name='title', bs=64)
dls.show_batch()
```

	user	title	rating
0	542	My Left Foot (1989)	4
1	422	Event Horizon (1997)	3
2	311	African Queen, The (1951)	4
3	595	Face/Off (1997)	4
4	617	Evil Dead II (1987)	1
5	158	Jurassic Park (1993)	5
6	836	Chasing Amy (1997)	3
7	474	Emma (1996)	3
8	466	Jackie Chan's First Strike (1996)	3
9	554	Scream (1996)	3

To represent collaborative filtering in PyTorch we can't just use the crosstab representation directly, especially if we want it to fit into our deep learning framework. We can represent our movie and user latent factor tables as simple matrices:

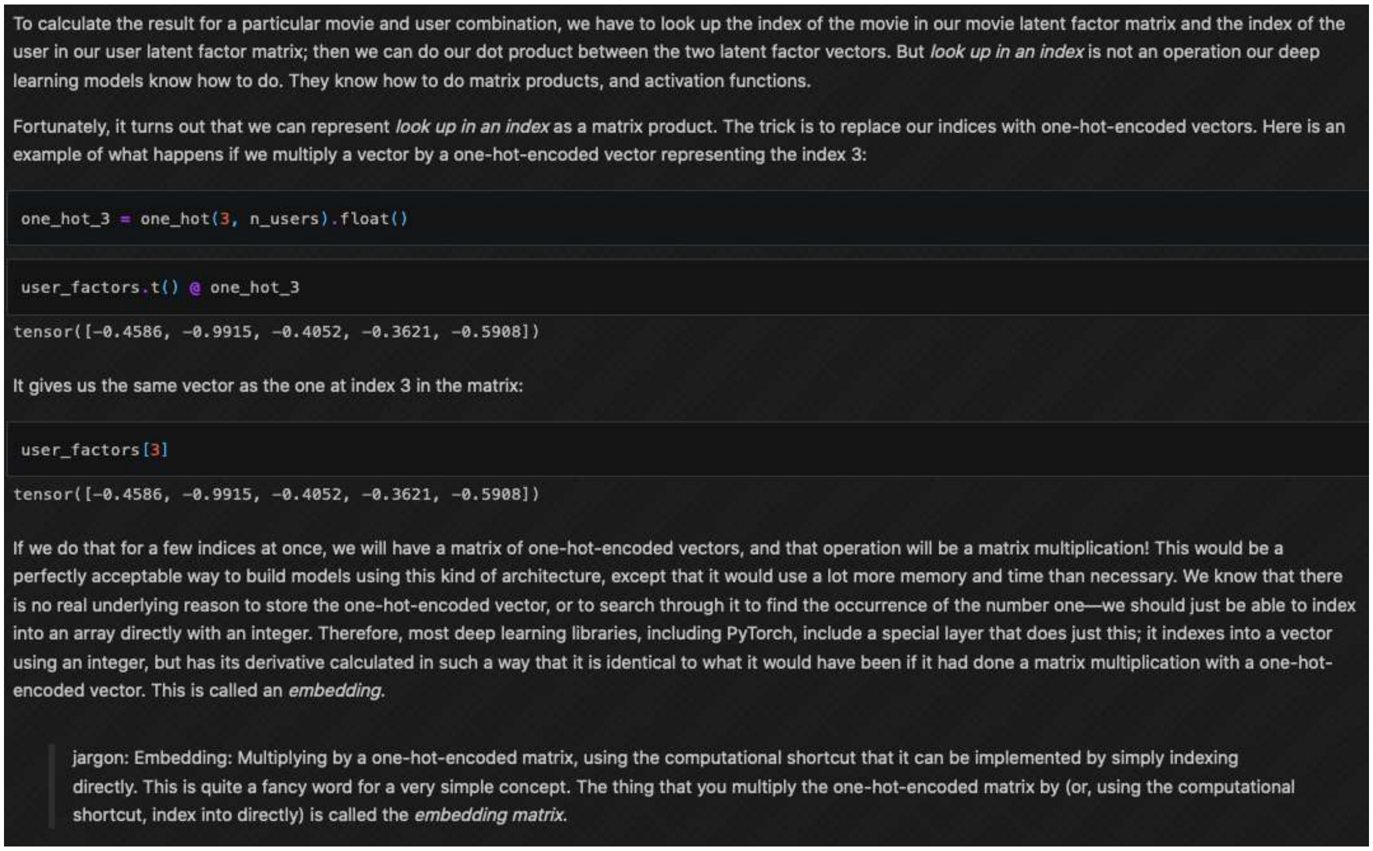
```
dls.classes
```

```
n_users=len(dls.classes['user'])
n_movies=len(dls.classes['title'])
n_factors=5 #latent factors

user_factors=torch.randn(n_users,n_factors)
movie_factors=torch.randn(n_movies,n_factors)
user_factors,movie_factors
```

```
(tensor([[-1.0827,  0.2138,  0.9310, -0.2739, -0.4359],
         [-0.5195,  0.7613, -0.4365,  0.1365,  1.3300],
         [-1.2804,  0.0705,  0.6489, -1.2110,  1.8266],
         ...,
         [ 0.8009, -0.4734, -0.8962, -0.7348, -0.0246],
         [ 0.3354, -0.8262, -0.1541,  0.4699,  0.4873],
         [ 2.4054, -0.2156, -1.4126, -0.2467,  1.0571]]),
 tensor([[-0.3978,  0.4563,  1.2301,  0.3745,  0.9689],
         [-1.1836, -0.5818, -0.5587, -0.4316,  0.2128],
         [ 0.0420,  1.3201, -0.7999,  1.1123, -0.7585],
         ...,
         [ 2.4743,  1.3068,  0.4540,  0.6958,  0.5228],
         [ 2.3970, -0.2559, -1.7196,  1.0440, -0.2662],
         [ 0.2786, -0.6593,  0.5260, -0.3416, -1.3938]]))
```

Embedding


To calculate the result for a particular movie and user combination, we have to look up the index of the movie in our movie latent factor matrix and the index of the user in our user latent factor matrix; then we can do our dot product between the two latent factor vectors. But *look up in an index* is not an operation our deep learning models know how to do. They know how to do matrix products, and activation functions.

Fortunately, it turns out that we can represent *look up in an index* as a matrix product. The trick is to replace our indices with one-hot-encoded vectors. Here is an example of what happens if we multiply a vector by a one-hot-encoded vector representing the index 3:

```
one_hot_3 = one_hot(3, n_users).float()
```

```
user_factors.t() @ one_hot_3
```

```
tensor([-0.4586, -0.9915, -0.4052, -0.3621, -0.5908])
```

It gives us the same vector as the one at index 3 in the matrix:

```
user_factors[3]
```

```
tensor([-0.4586, -0.9915, -0.4052, -0.3621, -0.5908])
```

If we do that for a few indices at once, we will have a matrix of one-hot-encoded vectors, and that operation will be a matrix multiplication! This would be a perfectly acceptable way to build models using this kind of architecture, except that it would use a lot more memory and time than necessary. We know that there is no real underlying reason to store the one-hot-encoded vector, or to search through it to find the occurrence of the number one—we should just be able to index into an array directly with an integer. Therefore, most deep learning libraries, including PyTorch, include a special layer that does just this; it indexes into a vector using an integer, but has its derivative calculated in such a way that it is identical to what it would have been if it had done a matrix multiplication with a one-hot-encoded vector. This is called an *embedding*.

> jargon: Embedding: Multiplying by a one-hot-encoded matrix, using the computational shortcut that it can be implemented by simply indexing directly. This is quite a fancy word for a very simple concept. The thing that you multiply the one-hot-encoded matrix by (or, using the computational shortcut, index into directly) is called the *embedding matrix*.

How Things Work

- In computer vision, we have a very easy way to get all the information of a pixel through its RGB values: each pixel in a colored image is represented by three numbers. Those three numbers give us the redness, the greenness and the blueness, which is enough to get our model to work afterward.
- For the problem at hand, we don't have the same easy way to characterize a user or a movie. There are probably relations with genres: if a given user likes romance, they are likely to give higher scores to romance movies. Other factors might be whether the movie is more action-oriented versus heavy on dialogue, or the presence of a specific actor that a user might particularly like.
- **How do we determine numbers to characterize those? The answer is, we don't. We will let our model learn them. By analyzing the existing relations between users and movies, our model can figure out itself the features that seem important or not.**
- **This is what embeddings are. We will attribute to each of our users and each of our movies a random vector of a certain length (here, n_factors=5), and we will make those learnable parameters. That means that at each step, when we compute the loss by comparing our predictions to our targets, we will compute the gradients of the loss with respect to those embedding vectors and update them with the rules of SGD (or another optimizer).**

At the beginning, those numbers don't mean anything since we have chosen them randomly, but by the end of training, they will. By learning on existing data about the relations between users and movies, without having any other information, we will see that they still get some important features, and can isolate blockbusters from independent cinema, action movies from romance, and so on.

We are now in a position that we can create our whole model from scratch.

Colaborative Filtering From Scratch

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

```
(torch.Size([64, 2]), torch.Size([64, 1]))
```

```
class DotProduct(Module):
    def __init__(self, n_users, n_movies, n_factors):
        self.user_factors = Embedding(n_users, n_factors)
        self.movie_factors = Embedding(n_movies, n_factors)

    def forward(self, x):
        users = self.user_factors(x[:,0])
        movies = self.movie_factors(x[:,1])
        return (users * movies).sum(dim=1)
```

Now that we have defined our architecture, and created our parameter matrices, we need to create a Learner to optimize our model. In the past we have used special functions, such as vision_learner, which set up everything for us for a particular application. Since we are doing things from scratch here, we will use the plain Learner class:

```
model=DotProduct(n_users,n_movies,50)
learn=Learner(dls,model,loss_func=MSELossFlat()) # GENERAL LEARNER
```

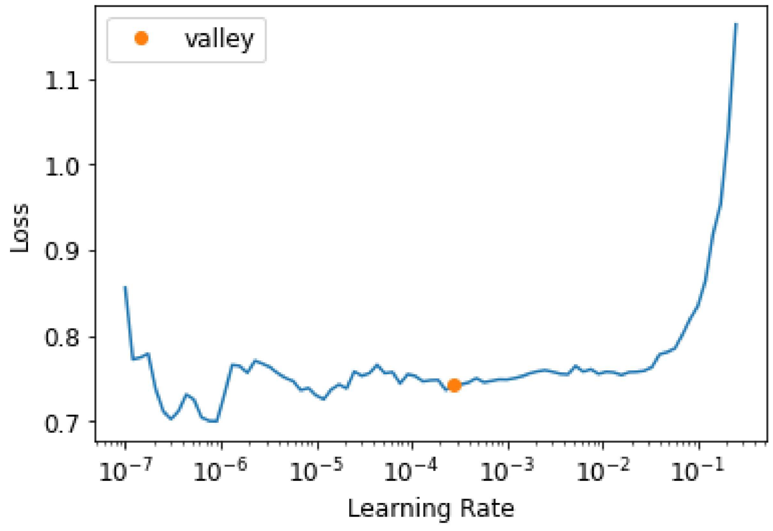
```
learn.fit_one_cycle(5,5e-3)
```

epoch	train_loss	valid_loss	time
0	1.344786	1.279100	00:12
1	1.093332	1.109981	00:11
2	0.958258	0.990199	00:21
3	0.814234	0.894916	00:10
4	0.780714	0.882022	00:10

- The first thing we can do to make this model a little bit better is to force those predictions to be between 0 and 5. For this, we just need to use sigmoid_range
- One thing we discovered empirically is that it's better to have the range go a little bit over 5, so we use (0, 5.5):

```
learn.lr_find()
```

```
SuggestedLRs(valley=0.0002754228771664202)
```



```
class DotProduct(Module):
    def __init__(self, n_users, n_movies, n_factors, y_range=(0,5.5)):
        self.user_factors = Embedding(n_users, n_factors)
        self.movie_factors = Embedding(n_movies, n_factors)
        self.y_range = y_range

    def forward(self, x):
        users = self.user_factors(x[:,0])
        movies = self.movie_factors(x[:,1])
        return sigmoid_range((users * movies).sum(dim=1), *self.y_range)
```

```
model=DotProduct(n_users,n_movies,50)
learn=Learner(dls,model,loss_func=MSELossFlat())
```

```
learn.fit_one_cycle(6,0.5e-3) # as per suggested loss
```

```
epoch  train_loss  valid_loss  time
```

Method did'nt worked well

Problem

- Model not knows if a certain user predicts everything bad
- or if a certain movie is generally very good or very bad

Solution

- add a bias

```
class DotProductBias(Module):
    def __init__(self, n_users, n_movies, n_factors, y_range=(0,5.5)):
        self.user_factors = Embedding(n_users, n_factors)
        self.user_bias = Embedding(n_users, 1)
        self.movie_factors = Embedding(n_movies, n_factors)
        self.movie_bias = Embedding(n_movies, 1)
        self.y_range = y_range

    def forward(self, x):
        users = self.user_factors(x[:,0])
        movies = self.movie_factors(x[:,1])
        res = (users * movies).sum(dim=1, keepdim=True)
        res += self.user_bias(x[:,0]) + self.movie_bias(x[:,1])
        return sigmoid_range(res, *self.y_range)
```

```
model = DotProductBias(n_users, n_movies, 50)
learn = Learner(dls, model, loss_func=MSELossFlat())
learn.fit_one_cycle(5, 5e-3)
```

epoch	train_loss	valid_loss	time
0	0.930319	0.935421	00:10
1	0.846721	0.863953	00:10
2	0.596207	0.867425	00:10
3	0.397392	0.889569	00:10
4	0.304442	0.896408	00:10

Overfitting Occured

Instead of being better, it ends up being worse (at least at the end of training). Why is that? If we look at both trainings carefully, we can see the validation loss stopped improving in the middle and started to get worse. As we've seen, this is a clear indication of overfitting. In this case, there is no way to use data augmentation, so we will have to use another regularization technique. One approach that can be helpful is weight decay.

Data Augmentation cant be done on this text based data

Problem : Overfitting

- Make it good by Reducing The Capacity of the Model
- means how much space it has to find the answers
- so its kind of memorizing stuff
- **By Reducing Number of Parameters , but a more better way exists - Weight Decay**

Solution : Weight Decay

- one liner -> decrease weights so it will fit less on trainnign set

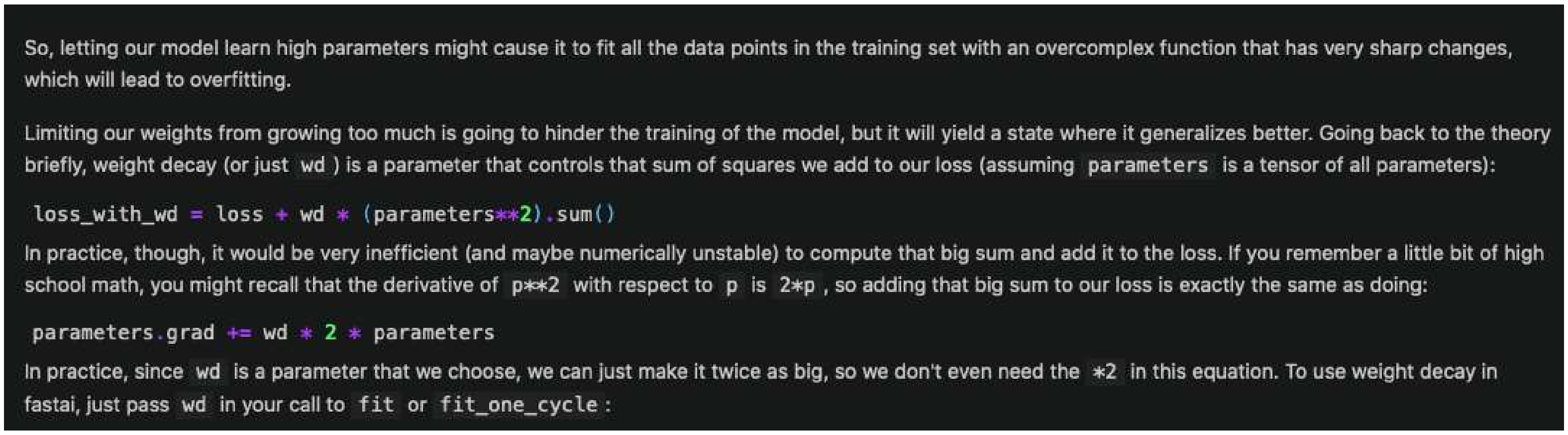
Weight decay, or L2 regularization, consists in adding to your loss function the sum of all the weights squared. Why do that? Because when we compute the gradients, it will add a contribution to them that will encourage the weights to be as small as possible.

Why would it prevent overfitting? The idea is that the larger the coefficients are, the sharper canyons we will have in the loss function.

So, letting our model learn high parameters might cause it to fit all the data points in the training set with an overcomplex function that has very sharp changes, which will lead to overfitting.

Limiting our weights from growing too much is going to hinder the training of the model, but it will yield a state where it generalizes better. Going back to the theory briefly, weight decay (or just wd) is a parameter that controls that sum of squares we add to our loss (assuming parameters is a tensor of all parameters):

https://github.com/fastai/fastbook/blob/master/08_collab.ipynb


So, letting our model learn high parameters might cause it to fit all the data points in the training set with an overcomplex function that has very sharp changes, which will lead to overfitting.

Limiting our weights from growing too much is going to hinder the training of the model, but it will yield a state where it generalizes better. Going back to the theory briefly, weight decay (or just `wd`) is a parameter that controls that sum of squares we add to our loss (assuming `parameters` is a tensor of all parameters):

```
loss_with_wd = loss + wd * (parameters**2).sum()
```

In practice, though, it would be very inefficient (and maybe numerically unstable) to compute that big sum and add it to the loss. If you remember a little bit of high school math, you might recall that the derivative of `p**2` with respect to `p` is `2*p`, so adding that big sum to our loss is exactly the same as doing:

```
parameters.grad += wd * 2 * parameters
```

In practice, since `wd` is a parameter that we choose, we can just make it twice as big, so we don't even need the `*2` in this equation. To use weight decay in fastai, just pass `wd` in your call to `fit` or `fit_one_cycle`:

```
model= DotProductBias(n_users,n_movies,50)
learn=Learner(dls,model,loss_func=MSELossFlat())
```

```
learn.fit_one_cycle(6,5e-3,wd=0.1)# wd-> hyperparameter
```

epoch	train_loss	valid_loss	time
0	0.984680	0.958121	00:11
1	0.860098	0.888018	00:11
2	0.779933	0.845343	00:10
3	0.618376	0.829547	00:10
4	0.510920	0.827281	00:10
5	0.419859	0.828673	00:10

Results

- Training Loss Goes Up -> as weights are decreased and now its more smooth , not fitting each point
- Where Valid loss has decreased so a SUCCESS it is

Creating Our Own Embedding Module

returns a tensor of passed size between random normally distributed random numbers with mean=0 and deviation=0.1

```
def create_params(size):
  return nn.Parameter(torch.zeros(*size).normal_(0,0.1))
```

```
class DotProductBias(Module):
    def __init__(self, n_users, n_movies, n_factors, y_range=(0,5.5)):
        self.user_factors = create_params([n_users, n_factors])
        self.user_bias = create_params([n_users])
        self.movie_factors = create_params([n_movies, n_factors])
        self.movie_bias = create_params([n_movies])
        self.y_range = y_range

    def forward(self, x):
        users = self.user_factors[x[:,0]]
        movies = self.movie_factors[x[:,1]]
        res = (users*movies).sum(dim=1)
        res += self.user_bias[x[:,0]] + self.movie_bias[x[:,1]]
        return sigmoid_range(res, *self.y_range)
```

```
model = DotProductBias(n_users, n_movies, 50)
learn = Learner(dls, model, loss_func=MSELossFlat())
learn.fit_one_cycle(8, 5e-3, wd=0.2)
```

epoch	train_loss	valid_loss	time
0	1.219162	1.194557	00:10
1	0.866192	0.895972	00:10
2	0.827364	0.874542	00:10
3	0.764316	0.856612	00:10
4	0.699742	0.841359	00:16
5	0.637247	0.831538	00:14
6	0.552374	0.827953	00:10
7	0.515929	0.828093	00:10

So Embedding Layer is nothing fancy but just indexing into an array for dot product - basically finding missing places in the matrix for recommendation **Colaborative Filtering Model**

Interpreting What is Learned

Embeddings and Biases

Our model is already useful, in that it can provide us with movie recommendations for our users—but it is also interesting to see what parameters it has discovered. The easiest to interpret are the biases. Here are the movies with the lowest values in the bias vector

Least Liked Movies By People

- recommended by self learned parameter Bias

```
movie_bias = learn.model.movie_bias.squeeze()
idxs = movie_bias.argsort()[:10]
[dls.classes['title'][i] for i in idxs]

# least liked
```

```
['Children of the Corn: The Gathering (1996)',
 'Lawnmower Man 2: Beyond Cyberspace (1996)',
 'Mortal Kombat: Annihilation (1997)',
 'Crow: City of Angels, The (1996)',
 'Home Alone 3 (1997)',
 'Robocop 3 (1993)',
 'Amityville 3-D (1983)',
 "Joe's Apartment (1996)",
 'Beautician and the Beast, The (1997)',
 'Kansas City (1996)']
```

Think about what this means. What it's saying is that for each of these movies, even when a user is very well matched to its latent factors (which, as we will see in a moment, tend to represent things like level of action, age of movie, and so forth), they still generally don't like it. We could have simply sorted the movies directly by their average rating, but looking at the learned bias tells us something much more interesting. It tells us not just whether a movie is of a kind that people tend not to enjoy watching, but that people tend not to like watching it even if it is of a kind that they would otherwise enjoy! By the same token, here are the movies with the highest bias

```
idxs = movie_bias.argsort(descending=True)[:10]
[dls.classes['title'][i] for i in idxs]
```

```
['L.A. Confidential (1997)',
 'Shawshank Redemption, The (1994)',
 'Titanic (1997)',
 'Silence of the Lambs, The (1991)',
 'Rear Window (1954)',
 "Schindler's List (1993)",
 'Star Wars (1977)',
 'Wrong Trousers, The (1993)',
 'Close Shave, A (1995)',
 'Good Will Hunting (1997)']
```

So, for instance, even if you don't normally enjoy detective movies, you might enjoy LA Confidential!

It is not quite so easy to directly interpret the embedding matrices. There are just too many factors for a human to look at. But there is a technique that can pull out the most important underlying directions in such a matrix, called principal component analysis (PCA).

Now Using Fastai To Build The Same Model Again

```
learn=collab_learner(dls,n_factors=50,y_range=(0,5.5))
```

```
learn.fit_one_cycle(5,5e-3,wd=0.1)
```

epoch	train_loss	valid_loss	time
0	0.954384	0.938620	00:10
1	0.838243	0.877727	00:10
2	0.738506	0.832984	00:10
3	0.584094	0.821634	00:10
4	0.499379	0.822265	00:10

we got same results here too - means our own model was written correct

Name of Layers

```
learn.model
```

```
EmbeddingDotBias(
  (u_weight): Embedding(944, 50)
  (i_weight): Embedding(1665, 50)
  (u_bias): Embedding(944, 1)
  (i_bias): Embedding(1665, 1)
)
```

using this bias to interpret ot get some useful results

```
movie_bias=learn.model.i_bias.weight.squeeze()
idxs=movie_bias.argsort(descending=True)[:10]
[dls.classes['title'][i] for i in idxs]
```

```
['L.A. Confidential (1997)',
 'Silence of the Lambs, The (1991)',
 'Titanic (1997)',
 'Shawshank Redemption, The (1994)',
 "Schindler's List (1993)",
 'Good Will Hunting (1997)',
 'Star Wars (1977)',
 'Usual Suspects, The (1995)',
 'Rear Window (1954)',
 'Wrong Trousers, The (1993)']
```

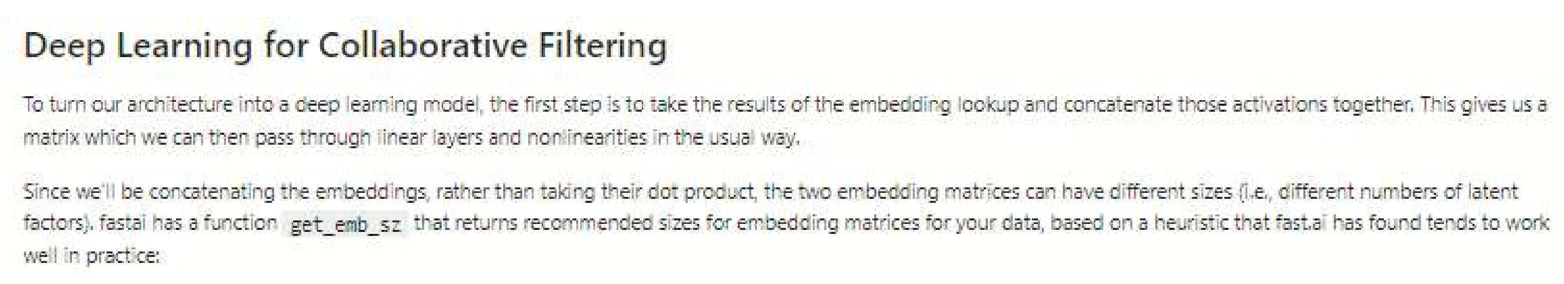
Embedding Distance

- Idea : pick a movie and find the to what movie it is the nearest
- formula ▸ (x**2+y**2)**1/2

```
movie_factors = learn.model.i_weight.weight
idx = dls.classes['title'].o2i['Silence of the Lambs, The (1991)']
distances = nn.CosineSimilarity(dim=1)(movie_factors, movie_factors[idx][None])
idx = distances.argsort(descending=True)[1]
dls.classes['title'][idx]
```

```
'Shawshank Redemption, The (1994)'
```

Using NN for Collaborative Filtering



Deep Learning for Collaborative Filtering

To turn our architecture into a deep learning model, the first step is to take the results of the embedding lookup and concatenate those activations together. This gives us a matrix which we can then pass through linear layers and nonlinearities in the usual way.

Since we'll be concatenating the embeddings, rather than taking their dot product, the two embedding matrices can have different sizes (i.e., different numbers of latent factors). fastai has a function `get_emb_sz` that returns recommended sizes for embedding matrices for your data, based on a heuristic that fast.ai has found tends to work well in practice:

```
embs = get_emb_sz(dls)
embs
```

```
[(944, 74), (1665, 102)]
```

```
class CollabNN(Module):
    def __init__(self, user_sz, item_sz, y_range=(0,5.5), n_act=100):
        self.user_factors = Embedding(*user_sz)
        self.item_factors = Embedding(*item_sz)
        self.layers = nn.Sequential(
            nn.Linear(user_sz[1]+item_sz[1], n_act),
            nn.ReLU(),
            nn.Linear(n_act, 1))
        self.y_range = y_range

    def forward(self, x):
        embs = self.user_factors(x[:,0]),self.item_factors(x[:,1])
        x = self.layers(torch.cat(embs, dim=1))
        return sigmoid_range(x, *self.y_range)
```

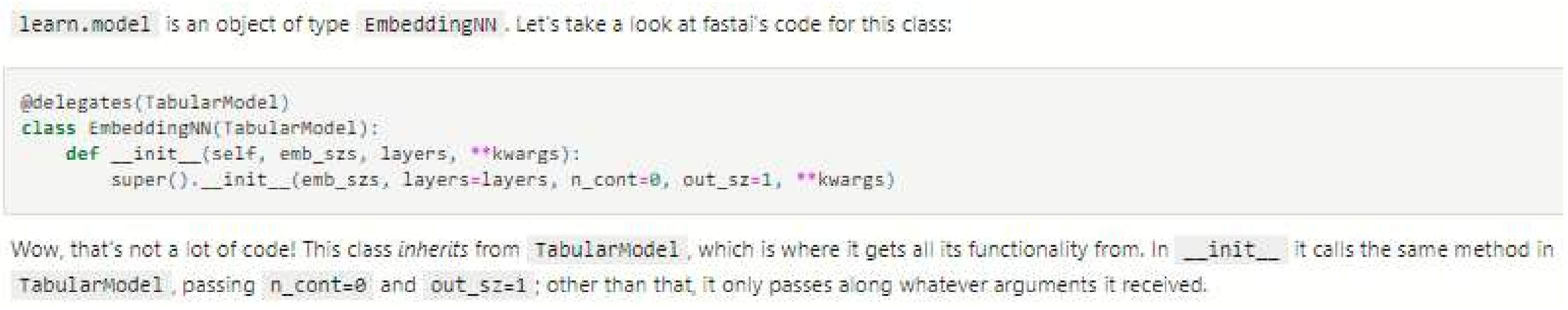
```
model = CollabNN(*embs)
```

fastai provides this model in fastai.collab if you pass use_nn=True in your call to collab_learner (including calling get_emb_sz for you), and it lets you easily create more layers. For instance, here we're creating two hidden layers, of size 100 and 50, respectively:

```
learn = collab_learner(dls, use_nn=True, y_range=(0, 5.5), layers=[100,50])
learn.fit_one_cycle(5, 5e-3, wd=0.1)
```

```
epoch  train_loss  valid_loss  time
    0    0.906480    0.967106  00:13
    1    0.895150    0.905932  00:17
    2    0.841435    0.882932  00:16
    3    0.831006    0.869680  00:13
    4    0.762019    0.872565  00:13
```

learn.model is an object of type EmbeddingNN. Let's take a look at fastai's code for this class:



`learn.model` is an object of type `EmbeddingNN`. Let's take a look at fastai's code for this class:

```
@delegates(TabularModel)
class EmbeddingNN(TabularModel):
    def __init__(self, emb_szs, layers, **kwargs):
        super().__init__(emb_szs, layers=layers, n_cont=0, out_sz=1, **kwargs)
```

Wow, that's not a lot of code! This class *inherits* from `TabularModel`, which is where it gets all its functionality from. In `__init__` it calls the same method in `TabularModel`, passing `n_cont=0` and `out_sz=1`; other than that, it only passes along whatever arguments it received.

✓ 1m 14s completed at 18:13