

```
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)
```

```
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'),Path('u.data'),Path('README'
```

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  

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 per page 2 10 100 1000 1900 2000

Fill these missing places

- recommend

		movieId														
		27	49	57	72	79	89	92	99	143	179	180	197	402	417	505
userId	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

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:

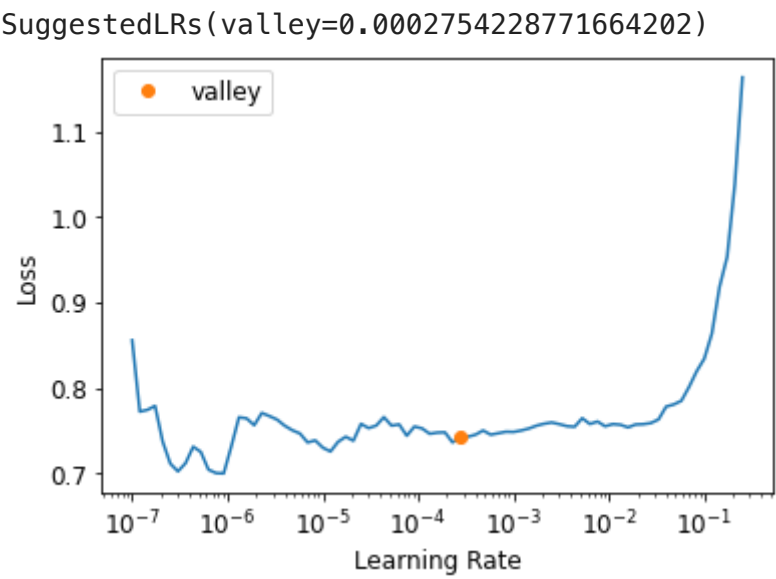
```
model=DotProduct(n_users,n_movies,50)
learn=Learner(dls,model,loss_func=MSELossFlat()) # GENERAL LEARNER
```

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

epoch	train_loss	valid_loss	time
0	1.344786	1.279100	00:11
1	1.093332	1.109981	00:17
2	0.958258	0.990199	00:08
3	0.814234	0.894916	00:08
4	0.780714	0.882022	00:08

- 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()
```



```
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
0	1.881979	1.851373	00:08
1	1.099632	1.122803	00:08
2	0.949700	0.982275	00:09
3	0.930679	0.954575	00:09
4	0.891365	0.945974	00:09
5	0.905293	0.944783	00:09

Method didn't work 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	00:09
1	0.846721	0.863953	00:09
2	0.596207	0.867425	00:09
3	0.397392	0.889569	00:09
4	0.304442	0.896408	00:09

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	0.956055	0.957230	00:09
1	0.862013	0.887571	00:09
2	0.752882	0.844432	00:09
3	0.634449	0.829812	00:09
4	0.495202	0.828099	00:09
5	0.413568	0.829460	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

