Collab

In [ ]:

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:

In [ ]:

```
embs = get_emb_sz(dls)
embs # [(944, 74), (1665, 102)]
# matlab 74 features for users, 102 features for movies
```

In [ ]:

In [ ]:

In [ ]:

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

In [ ]:

```
dls = CollabDataLoaders.from_csv(path/'ratings.csv')
learn = collab_learner(dls, y_range=(0.5,5.5))
learn.fine_tune(10)
# epoch train_loss      valid_loss      time
# 0     1.515542        1.374456        00:00
# epoch train_loss      valid_loss      time
# 0     1.377571        1.317657        00:00
# 1     1.276605        1.144093        00:00
# ..
# 8     0.599054        0.683095        00:00
# 9     0.612227        0.682993        00:00
```

In [ ]:

```
learn.show_results()
# userId        movieId rating  rating_pred
# 0     74.0    39.0    3.0     4.015666
# 1     39.0    56.0    5.0     4.028876
# ..
# 7     37.0    13.0    5.0     3.406052
# 8     51.0    43.0    5.0     3.713690
```

main thing to learn about users and movies are latent factors

we want to fill missing gaps

	movieId														
userId	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

Ratings df looks like

	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)

By default, DataLoaders takes the first column for the user, the second column for the item (item_name)- movies, and the third column for the ratings.

In [ ]:

```
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
```

In [ ]:

```
x,y = dls.one_batch()
# x has list of [userID, movieID], y is rating
```

force those predictions to be between 0 and 5. For this, we just need to use `sigmoid_range` , like in <>.

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)`

In [ ]:

```
# neeche dekhenge DotProduct kaisa hota hai
# basically it returns two embedding matrices
# model.user_factors is Embedding(944, 50) since 944 users
# model.movie_factors is Embedding(1665, 50) since 1665 users

model = DotProduct(n_users, n_movies, 50)
```

```
learn = Learner(dls, model, loss_func=MSELossFlat())
# or
learn = collab_learner(dls, n_factors=50, y_range=(0, 5.5))
learn.fit_one_cycle(5, 5e-3, wd=0.1) # wd is weight decay (optional but useful for regularisation)
```

In [ ]:
```
movie_bias = learn.model.movie_bias.squeeze()
idxs = movie_bias.argsort(descending=True)[:2]
[dls.classes['title'][i] for i in idxs]
# ['Titanic (1997)',
#  'Shawshank Redemption, The (1994)']
```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```
# when collablearner use kiya tab ka
learn.model
# EmbeddingDotBias(
#   (u_weight): Embedding(944, 50)
#   (i_weight): Embedding(1665, 50)
#   (u_bias): Embedding(944, 1)
#   (i_bias): Embedding(1665, 1)
# )
learn.model.i_bias.weight.squeeze().argsort(descending=True)
# gets movies with highest bias
# tensor([ 373, 1591,  374,  ..., 1649,  690,  644])

movie_bias = learn.model.i_bias.weight.squeeze()
idxs = movie_bias.argsort(descending=True)[:5]
[dls.classes['title'][i] for i in idxs] # gives 5 top movies, result may be different here
```

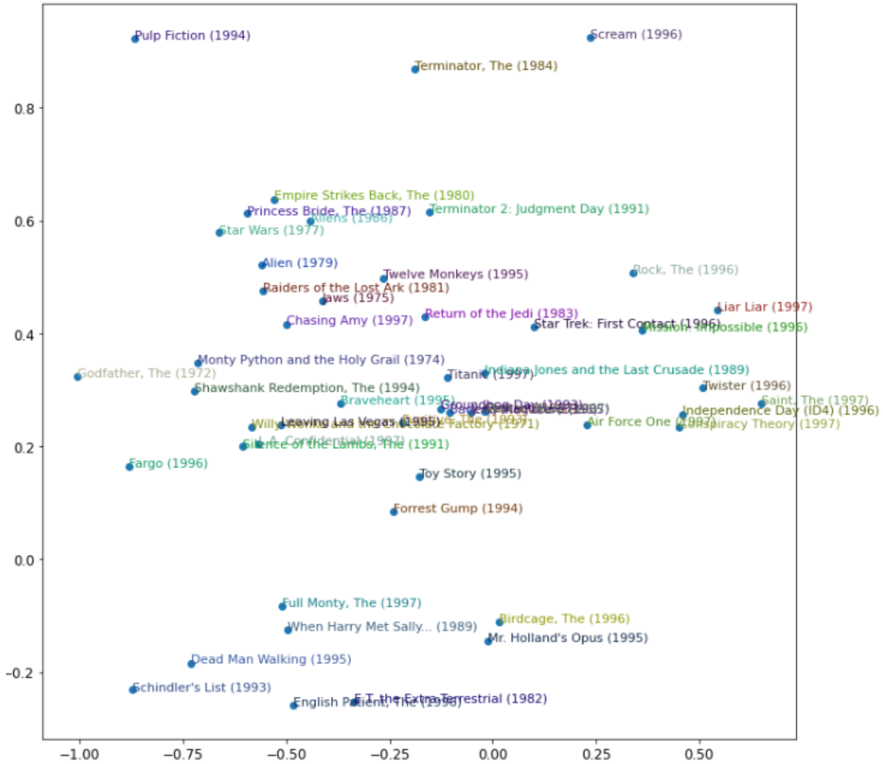
In [ ]:
```

```

shows what our movies look like **based on two of the strongest PCA components.**

In [ ]:
```
#hide_input
#id img_pca_movie
#caption Representation of movies based on two strongest PCA components
#alt Representation of movies based on two strongest PCA components
g = ratings.groupby('title')['rating'].count()
top_movies = g.sort_values(ascending=False).index.values[:1000]
top_idxs = tensor([learn.dls.classes['title'].o2i[m] for m in top_movies])
```

```python
movie_w = learn.model.movie_factors[top_idxs].cpu().detach()
movie_pca = movie_w.pca(3)
fac0,fac1,fac2 = movie_pca.t()
idxs = list(range(50))
X = fac0[idxs]
Y = fac2[idxs]
plt.figure(figsize=(12,12))
plt.scatter(X, Y)
for i, x, y in zip(top_movies[idxs], X, Y):
    plt.text(x,y,i, color=np.random.rand(3)*0.7, fontsize=11)
plt.show()
```



model seems to have discovered a concept of classic versus pop culture movies, or perhaps it is critically acclaimed that is represented here.

In [ ]:

In [ ]:

DL use

In [ ]:

```
class CollabNN(Module):
    def __init__(self, user_sz, item_sz, y_range=(0,5.5), n_act=100):
        # seems like * preserves parameter passed ka type
        # jaise yaha sz is still the shape (type = shape)
        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)
```

CollabNN creates our Embedding layers in the same way as previous classes in this chapter, except that we now use the embs sizes. self.layers is identical to the mini-neural net we created in chapter_mnist_basics for MNIST. Then, in forward, we apply the embeddings, concatenate the results, and pass this through the mini-neural net. Finally, we apply sigmoid_range as we have in previous models.

In [ ]:

```
model = CollabNN(*embs)
learn = Learner(dls, model, loss_func=MSELossFlat())
learn.fit_one_cycle(5, 5e-3, wd=0.01)
# epoch train_loss      valid_loss      time
# 0     0.946587        0.959500        00:11
# 1     0.914668        0.900304        00:11
# 2     0.850525        0.884657        00:11
# 3     0.814607        0.877216        00:11
# 4     0.769289        0.879043        00:11
```

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:

In [ ]:

```
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     1.006306        0.993857        00:14
# 1     0.878367        0.927818        00:15
# 2     0.883003        0.899196        00:15
# 3     0.813805        0.864728        00:15
```