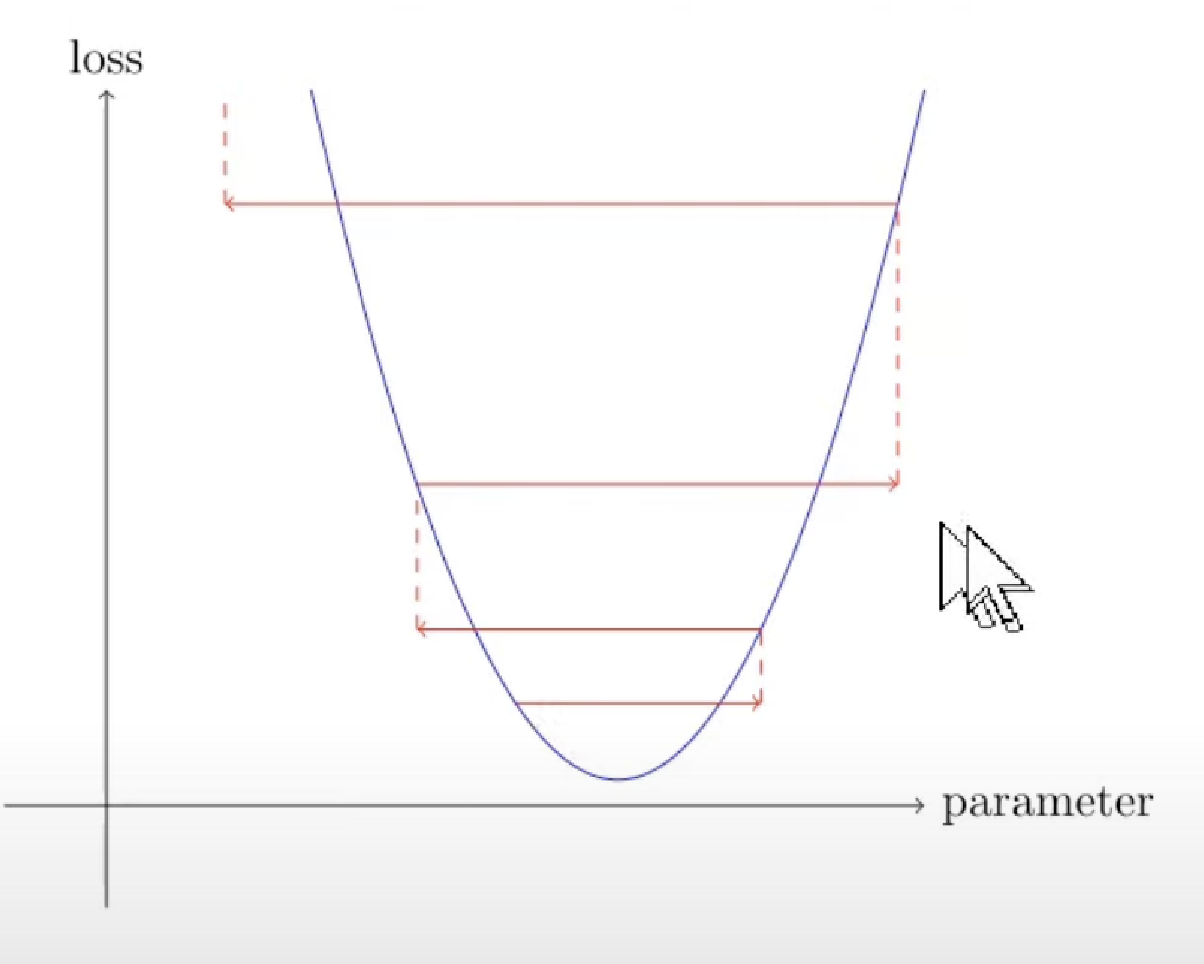
Lesson 6

With a large number of classes, using the confusion matrix `.plot_confusion_matrix` can be confusing and hard to read. Instead, the `.most_confused` method can be used.

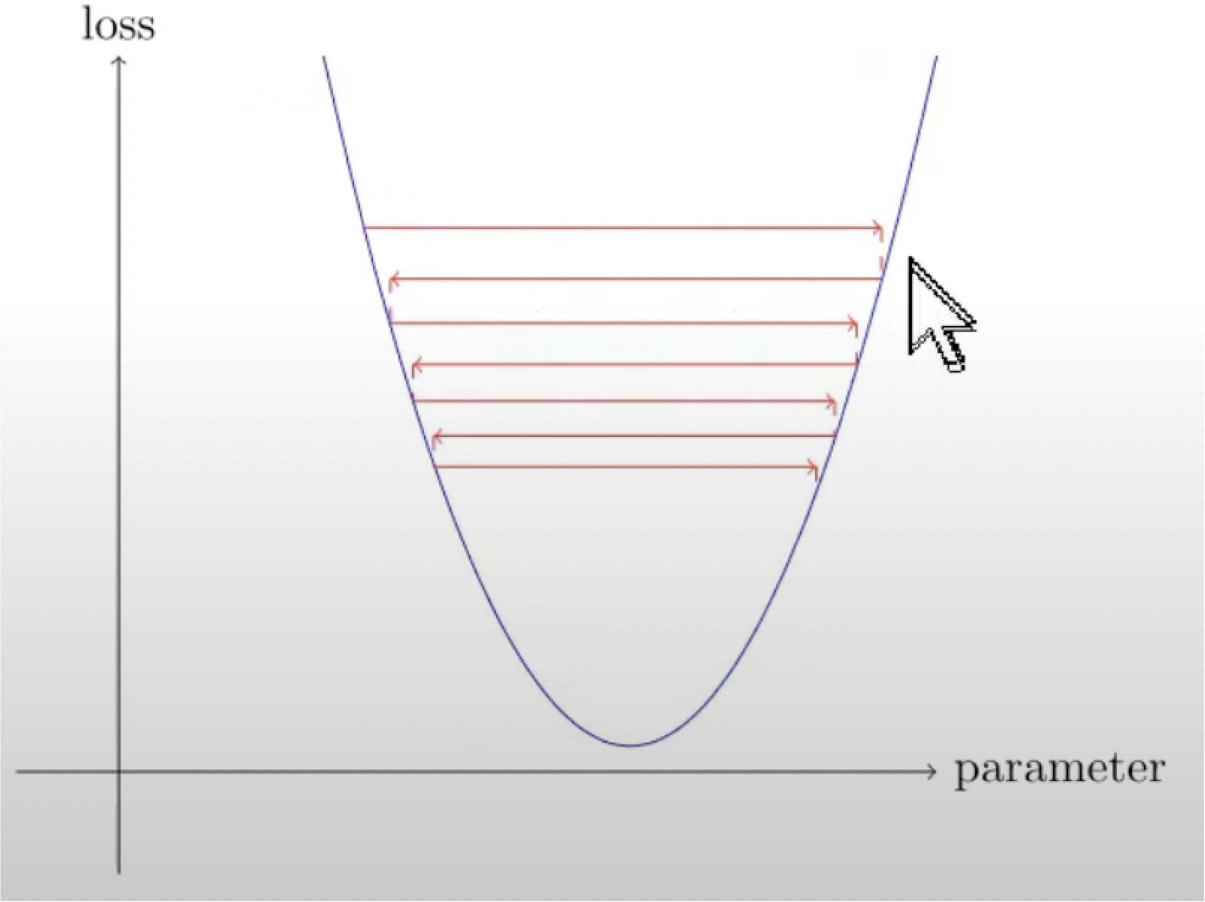
```
interp = ClassificationInterpretation.from_learner(learn)
#Confusion Matrix
interp.plot_confusion_matrix(figsize=(12.12), dpi=60)
#Most Confused
interp.most_confused(min_val=5)
```

We want to increase the `lr` as it speeds up the program.

Increasing the `lr` too much can drive us away from the minimus, increasing the loss.

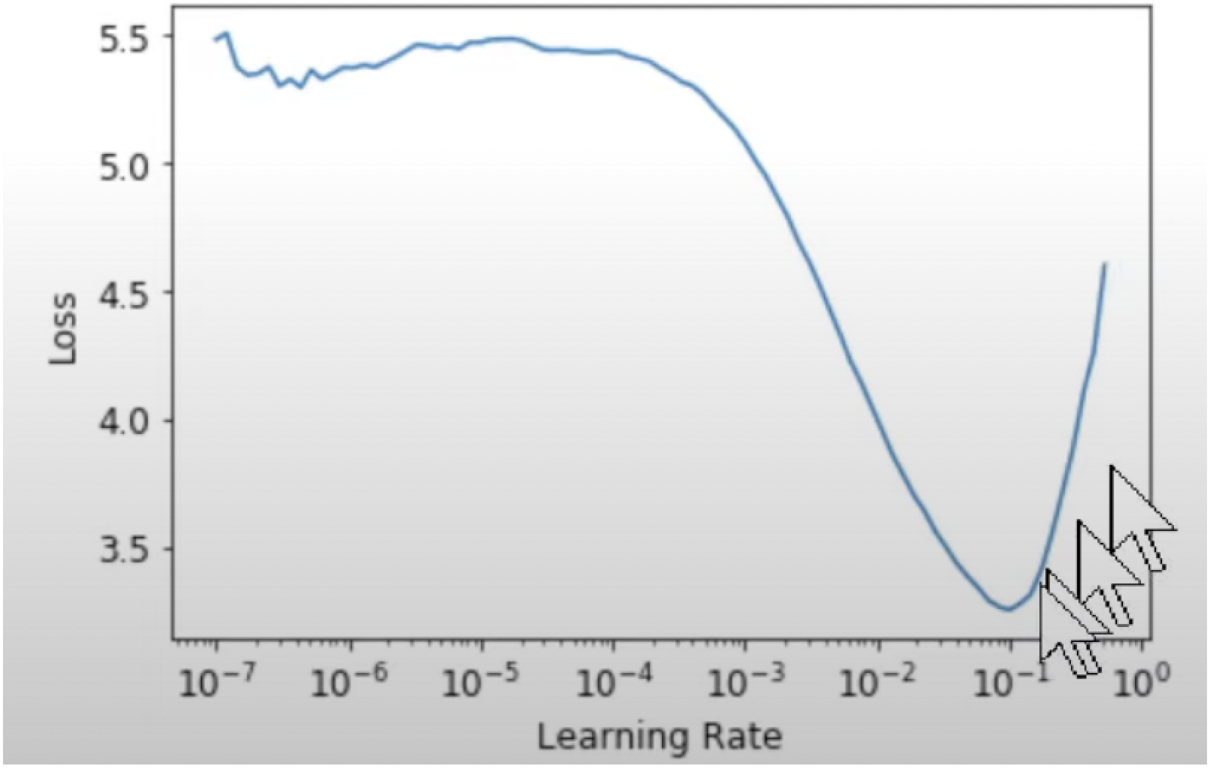


if the learning rate is too high, it might also just bounce around, instead of diverging.



Learning Rate Finder

- does the first mini-batch at a really low learning rate
- then increase the learning rate by a little bit (25%) in the next mini-batch
- then do the same
- and plot the loss for the same
- eventually we reach a point where there is a meaningful difference to the loss, it starts decreasing
- and then it reaches a point, where the loss starts increasing, the learning rate is too high



We want to select a learning rate where it's nice and steep and the learning rate is decreasing.

We want the steepest and not the minimum because at minimum the network is not learning anymore.

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
#learning rate finder
lr_min, lr_steep = learn.lr_find()
#print ideal values
print(f"Minimum/10: {lr_min:.2e}, steepest point: {lr_steep:.2e}")
CONSOLE: Minimum/10: 1.00e-02, steepest point: 3.03e-03
```

Selected on the basis of the above is `3e-3`.

Transfer Learning

- We throw away the last linear layer of weights and replace it with a new linear layer of random weights
- All of the prior layers have been trained to be good at image training (for example) tasks in general
- The last layer specializes it in our required dataset

`.fine_tune` is used by fastai to achieve transfer learning

- starts with `.freeze` : method which only the last few layers will get stepped by the optimizer, only their gradients will be calculated
- `fit_one_cycle`: steps only the randomly added layers' parameters for `freeze_epochs`
- all the layers except the last are the same as the pre-trained network, the last layer has been tuned for the task at hand
- closer you get to the right answer, the smaller you want the learning rate, so we divide the learning rate by two
- `.unfreeze` so that all the parameters can now be stepped
- `fit_one_cycle` steps all parameters for passed `epochs`

Replicating the `fine_tune` operation manually

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
#learn automatically freezes the layers for us
learn.fit_one_cycle(3, 3e-3)
learn.unfreeze()
#finding a new learning rate
learn.lr_find()
learn.fit_one_cycle(6, lr_max=1e-5)
```

But

- the later layers have not been trained much, and have to be specialized for our task, so they probably need higher learning rates
- the earlier layers have already been trained, so they don't need as high learning rates
- Therefore, different layers have to be trained at different learning rates

fastai lets you pass a `slice` where the earliest layer will have the first value and the last layer will have the last value with the middle layers having equal multiples of the difference. eg. `slice(1e-6,1e-4)` means that the first layer will have an `lr` of `1e-6` and the last will have `1e-4` while the once in the middle will have a multiple of the difference.

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fit_one_cycle(3, 3e-3)
learn.unfreeze()
learn.fit_one_cycle(12, lr_max=slice(1e-6,1e-4))
```

`fit_one_cycle` starts at a low `lr` and increases it for the first third batches, and then decreases for the rest of the batches, in contrast to `fit`.

Reason: Based on emperical reasearch of Leslie Smith to improve the speed and accuracy of neural networks.

A deeper architecture would be models with more pairs of non-linear and linear layers.

If you are getting out of memory error, you can use `.to_fp16()`

- numbers use half as many bits (half precision), so half as accurate
- numbers use less memory as a result
- results in 2-3 times speedup in NVIDIA GPUs from >2020

In this example, resnet50 runs in roughle the same time as what originally took resnet32 to run originally

```
from fastai.callback.fp16 import *
learn = cnn_learner(dls, resnet50, metrics=error_rate).to_fp16()
learn.fine_tune(6, freeze_epochs=3)
```

Note: bigger models don't always yield better neural networks, the above example will probably give a similar error rate as resnet34.

Chapter 6

`pandas` is a dataframe library

PyTorch has two classes for bringing training and validation datasets together:

`Dataset`: tuple of independent and dependent variable for a *single* item

`DataLoader`: iterator that provides a stream of mini-batches, where each mini-batch is a tuple of a batch of independent variables and dependent variables

fastai builds on PyTorch with:

`Datasets`: object that contains a training `Dataset` and a validation `Dataset`

`DataLoaders`: object that contains a training `DataLoader` and a validation `DataLoader`

`Datasets` automatically creates a random 80-20 training validation split from the given data

lambdas are not compatible with serialization, python doesn't like saving things with lambda, so use the more verbose approach if you want to export your `Learner.`

MultiCategoryBlock allows the categories to be read as a *one-hot encoded vector*. Its a vector of zeroes with a one for all categories which apply for that particular image.

- we didn't have to use it before because we only had one applicable category in the previous example
- the dependent variable can be checked using `.vocab`

```
idxs = torch.where(dsets.train[0][1]==1.)[0]
dsets.train.vocab[idxs]
CONSOLE: (#1) ['dog']
```

`.model` is a function that returns activations from the final layer. Calling `.shape` on it returns 64,20:

- batch size is 64 so the model function is going to predict the categories for 64 images at a time
- it is predicting 20 categories as a one-hot encoded vector

We can't directly use cross entropy loss because:

- `softmax` requires that activations add up to 1. Which might not happen because more than 1 categories might be true, and it will be greater than 1. Conversely, no category might be true, so, the activation might be less than 1.
- `nll_loss` returns the value of just one activation. Again, more than one categories might be present, or no categories might be present.

On the other hand, *binary cross entropy loss* which is just `mnist_loss` with `log` can be utilized for the problem due to broadcasting

```
def binary_cross_entropy(inputs, targets):
  inputs = inputs.sigmoid()
  return torch.where(targets==1, 1-inputs, inputs).log().mean()
```

Chapter 8

Building a recommendation engine

Depends on the concept of latent factors, which are a set of factors used to describe an item and the user. For example, a movie could have three latent factors: science fiction, action, and old movie. Then Last Skywalker's latent factors could be represented as:

```
last_skywalker = np.array([0.98, 0.9, -0.9])
#very sci-fi, very action, very not old
```

We could represent a user who likes modern sci-fi action movies as:

```
user = np.array([0.9, 0.8, -0.6])
```

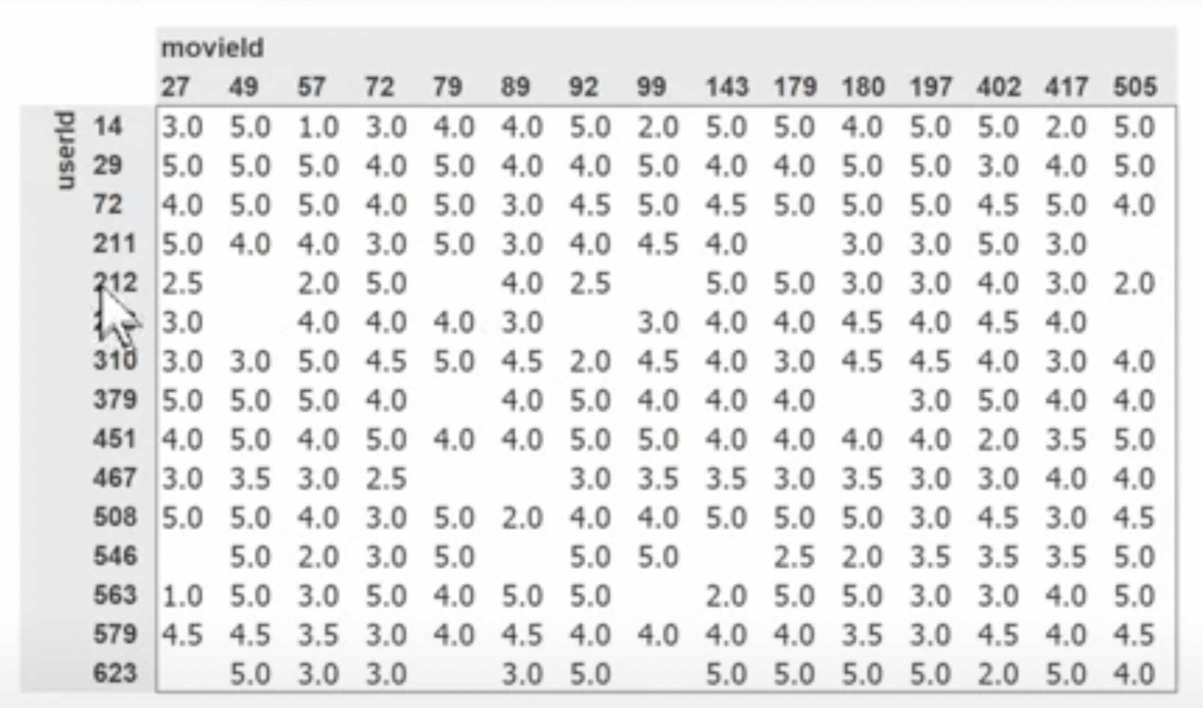
Now we can calculate the match between this combination with a dot product

```
(user*last_skywalker).sum()
CONSOLE: 2.142
```

On the other hand, Casablanca could be represented as:

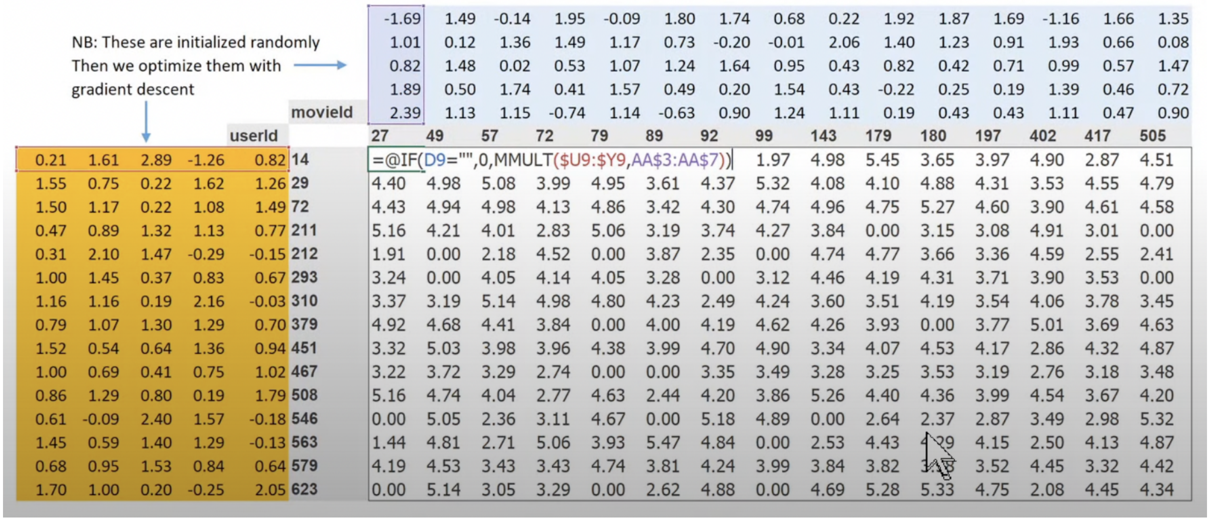
```
casablanca = np.array([-0.99, -0.3, 0,8])
(user*casablanca).sum()
CONSOLE: -1.611
```

Objective is to fill in the gaps, and then recommend movies with high predicted ratings



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
[illegible]	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

We assume an `n` number of latent factors each for users and movies. We initialize them with random values (weights). The dot products of the two sets give us the rating, so we can fit a model to match the ratings by calculating the loss by comparing the values with the rated values and doing SGD.



NB: These are initialized randomly
Then we optimize them with
gradient descent

movieId	27	49	57	72	79	89	92	99	143	179	180	197	402	417	505
	-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
	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	[illegible]	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

We have to take the dot product of specific movie with a specific user, so we can't do dot product directly.

We have to look up the index in the movie latent factors and look up the index in the user latent factors and then do the dot product.

We can solve this problem by replacing indices with *one hot encoded vectors*.

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

This creates a vector of length equal to n_users (944) with 0s throughtout and a single 1 and index 3. Now if we multiply it with `user_factors`:

```
user_factors.t() @ one_hot_3 # eq 1
# this above is something we know how to do SGD with
CONSOLE: tensor([-0.4586, -0.9915, -0.4052, -0.3621, -0.5908])
user_factors[3] # eq 2
CONSOLE: tensor([-0.4586, -0.9915, -0.4052, -0.3621, -0.5908])
```

it spits out the 4th index row of user_factors because everything else got multiplied by zero.

> Matrix multiplication with one-hot encoding is a way to index into an array

---

Its standard practice to tune the number of latent factors

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*.

> embedding is a computational shortcut that has the speed of an array lookup (like `eq 2`) but the gradient computational capabilities of a matrix multiplication (like `eq 1`)

When you call classes inheriting from PyTorch class `Module`, they call the `forward` method. That is a special method that contains the actual computation.

```
x,y = dls.one_batch()
x.shape
CONSOLE: torch.Size([64,2])
```

Because each batch has 64 items, and each item has two independent variables — the `user id` and the `movie id.`

Modifications to the model

- When you are doing regression, always use a range. It yields better results.
- Introducing bias, since some users rate movies high or low all the time or some movies are rated high or low by everybody