Fastai Lesson 5

2018年7月

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Lesson 5 的主要内容分为两个部分

第一部分，是介绍协同过滤的方法，以及协同过滤的实现。

第二部分，讲解梯度下降的过程，以及梯度下降的优化算法。

目录 Contents

协同过滤（Collaborative Filtering）

协同过滤（Collaborative Filtering），简单来说是利用某兴趣相投、拥有共同经验之群体的喜好来推荐用户感兴趣的信息，个人通过合作的机制给予信息相当程度的回应（如评分）并记录下来以达到过滤的目的进而帮助别人筛选信息，回应不一定局限于特别感兴趣的，特别不感兴趣信息的纪录也相当重要。

数据集来自Movielens，数据集内保存了大量的用户电影评分数据，可用于测试推荐算法。

http://files.grouplens.org/datasets/movielens/ml-latest-small.zip

	userId	movieId	rating	timestamp
0	1	31	2.5	1260759144
1	1	1029	3.0	1260759179
2	1	1061	3.0	1260759182
3	1	1129	2.0	1260759185
4	1	1172	4.0	1260759205

数据集中有多个文件，但是这堂课只需要用到rating.csv，其中储存了对应的用户ID对某一部电影（其ID与电影名称对应可在movies.csv中找到）的评分。

读取数据集后，从中选取评分数量最多的15个用户，以及被评分数量最多的15部电影用于分析，形成一个总表。

```
g=ratings.groupby('userId')['rating'].count()
topUsers=g.sort_values(ascending=False)[:15]

g=ratings.groupby('movieId')['rating'].count()
topMovies=g.sort_values(ascending=False)[:15]

top_r = ratings.join(topUsers, rsuffix='_r', how='inner', on='userId')
top_r = top_r.join(topMovies, rsuffix='_r', how='inner', on='movieId')

pd.crosstab(top_r.userId, top_r.movieId, top_r.rating, aggfunc=np.sum)
```

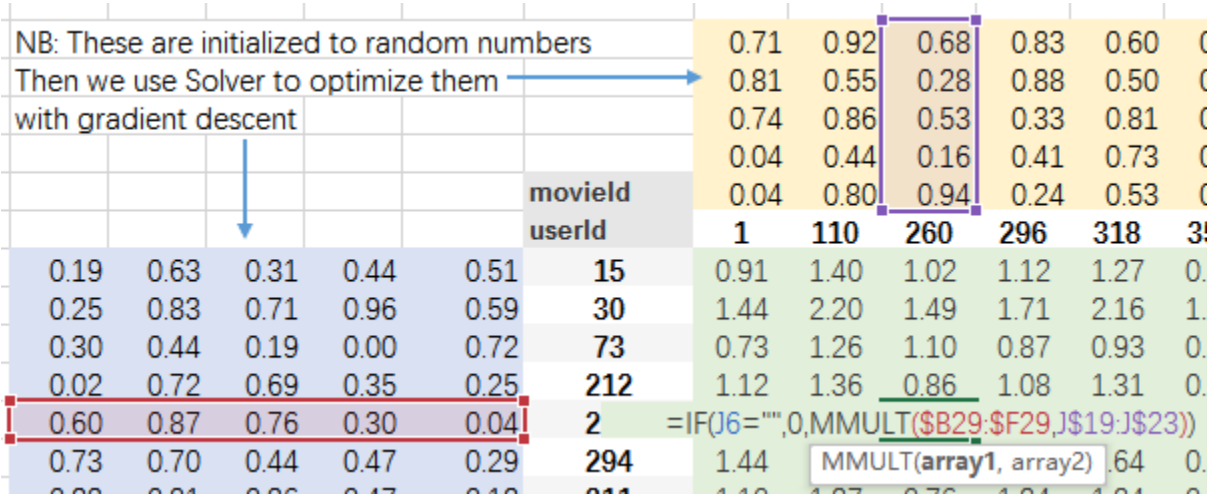
movieId	1	110	260	296	318	356	480	527	589	593	608	1196	1198	1270	2571
userId															
15	2.0	3.0	5.0	5.0	2.0	1.0	3.0	4.0	4.0	5.0	5.0	5.0	4.0	5.0	5.0
30	4.0	5.0	4.0	5.0	5.0	5.0	4.0	5.0	4.0	4.0	5.0	4.0	5.0	5.0	3.0
73	5.0	4.0	4.5	5.0	5.0	5.0	4.0	5.0	3.0	4.5	4.0	5.0	5.0	5.0	4.5
212	3.0	5.0	4.0	4.0	4.5	4.0	3.0	5.0	3.0	4.0	NaN	NaN	3.0	3.0	5.0
213	3.0	2.5	5.0	NaN	NaN	2.0	5.0	NaN	4.0	2.5	2.0	5.0	3.0	3.0	4.0
294	4.0	3.0	4.0	NaN	3.0	4.0	4.0	4.0	3.0	NaN	NaN	4.0	4.5	4.0	4.5
311	3.0	3.0	4.0	3.0	4.5	5.0	4.5	5.0	4.5	2.0	4.0	3.0	4.5	4.5	4.0
380	4.0	5.0	4.0	5.0	4.0	5.0	4.0	NaN	4.0	5.0	4.0	4.0	NaN	3.0	5.0
452	3.5	4.0	4.0	5.0	5.0	4.0	5.0	4.0	4.0	5.0	5.0	4.0	4.0	4.0	2.0
468	4.0	3.0	3.5	3.5	3.5	3.0	2.5	NaN	NaN	3.0	4.0	3.0	3.5	3.0	3.0
509	3.0	5.0	5.0	5.0	4.0	4.0	3.0	5.0	2.0	4.0	4.5	5.0	5.0	3.0	4.5
547	3.5	NaN	NaN	5.0	5.0	2.0	3.0	5.0	NaN	5.0	5.0	2.5	2.0	3.5	3.5
564	4.0	1.0	2.0	5.0	NaN	3.0	5.0	4.0	5.0	5.0	5.0	5.0	5.0	3.0	3.0
580	4.0	4.5	4.0	4.5	4.0	3.5	3.0	4.0	4.5	4.0	4.5	4.0	3.5	3.0	4.5
624	5.0	NaN	5.0	5.0	NaN	3.0	3.0	NaN	3.0	5.0	4.0	5.0	5.0	5.0	2.0

	movieId	1	110	260	296	318	356	480	527	589	593	608	1196	1198	1270	2571
userId	15	2	3	5	5	2	1	3	4	4	5	5	5	4	5	5
	30	4	5	4	5	5	5	4	5	4	4	5	4	5	5	3
	73	5	4	4.5	5	5	5	4	5	3	4.5	4	5	5	5	4.5
	212	3	5	4	4	4.5	4	3	5	3	4			3	3	5
	213	3	2.5	5			2	5		4	2.5	2	5	3	3	4
	294	4	3	4		3	4	4	4	3			4	4.5	4	4.5
	311	3	3	4	3	4.5	5	4.5	5	4.5	2	4	3	4.5	4.5	4
	380	4	5	4	5	4	5	4		4	5	4	4		3	5
	452	3.5	4	4	5	5	4	5	4	4	5	5	4	4	4	2
	468	4	3	3.5	3.5	3.5	3	2.5			3	4	3	3.5	3	3
	509	3	5	5	5	4	4	3	5	2	4	4.5	5	5	3	4.5
	547	3.5			5	5	2	3	5		5	5	2.5	2	3.5	3.5
	564	4	1	2	5		3	5	4	5	5	5	5	5	3	3
	580	4	4.5	4	4.5	4	3.5	3	4	4.5	4	4.5	4	3.5	3	4.5
	624	5		5	5		3	3		3	5	4	5	5	5	2

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

	movieId	1	110	260	296	318	356	480	527	589	593	608	1196	1198	1270	2571
		0.71	0.92	0.68	0.83	0.60	0.18	0.26	0.91	0.99	0.52	0.91	0.53	0.23	0.75	0.43
		0.81	0.55	0.28	0.88	0.50	0.31	0.08	0.47	0.94	0.70	0.11	0.87	0.20	0.47	0.81
		0.74	0.86	0.53	0.33	0.81	0.68	0.92	0.61	0.46	0.64	0.24	0.25	0.83	0.05	0.17
		0.04	0.44	0.16	0.41	0.73	0.39	0.29	0.94	0.12	0.67	0.54	0.57	0.53	0.91	0.30
		0.04	0.80	0.94	0.24	0.53	0.09	0.74	0.13	0.39	0.44	0.81	0.80	0.23	0.59	0.29
	movieId	1	110	260	296	318	356	480	527	589	593	608	1196	1198	1270	2571
	userId															
0.19	15	0.91	1.40	1.02	1.12	1.27	0.66	0.89	1.13	1.18	1.27	0.96	1.39	0.77	1.15	0.92
0.25	30	1.44	2.20	1.49	1.71	2.16	1.22	1.49	2.04	1.71	2.08	1.48	2.06	1.46	1.84	1.36
0.30	73	0.73	1.26	1.10	0.87	0.93	0.38	0.82	0.68	1.07	0.90	0.95	1.16	0.47	0.86	0.72
0.02	212	1.12	1.36	0.86	1.08	1.31	0.85	0.97	1.14	1.15	1.31	0.00	0.00	0.96	0.85	0.89
0.60	213	1.71	1.86	1.14	0.00	0.00	1.02	1.03	0.00	1.82	1.64	1.01	1.47	1.11	1.19	1.20
0.73	294	1.44	1.88	1.27	0.00	1.64	0.86	0.99	1.74	1.76	0.00	0.00	1.61	0.98	1.50	1.18
0.23	311	1.10	1.27	0.76	1.24	1.24	0.73	0.67	1.26	1.26	1.29	0.73	1.28	0.79	1.07	0.99
0.68	380	1.43	2.30	1.67	1.98	2.09	0.96	1.24	0.00	2.02	2.07	1.86	2.32	0.00	2.22	1.55
0.81	452	1.52	1.88	1.28	1.41	1.55	0.90	1.15	1.59	1.66	1.42	1.20	1.22	1.05	1.09	0.92
0.70	468	1.70	2.35	1.49	1.84	2.24	1.30	1.49	0.00	0.00	2.08	1.59	1.83	1.56	1.81	1.29
0.50	509	1.16	2.10	1.65	1.28	1.79	0.92	1.56	1.55	1.47	1.59	1.56	1.60	1.20	1.43	0.93
0.16	547	0.90	0.00	0.00	1.00	1.68	0.88	1.60	1.27	0.00	1.47	1.39	1.51	1.17	1.26	0.80
0.91	564	1.82	2.06	1.32	1.78	0.00	1.01	1.09	1.87	2.01	1.68	1.26	1.51	1.12	1.32	1.21
0.55	580	1.43	2.36	1.67	1.73	2.21	1.17	1.57	2.15	1.78	2.05	1.76	2.03	1.45	1.96	1.29
0.94	624	1.22	0.00	1.25	1.29	0.00	0.65	0.95	0.00	1.51	1.20	1.31	1.15	0.79	1.16	0.81

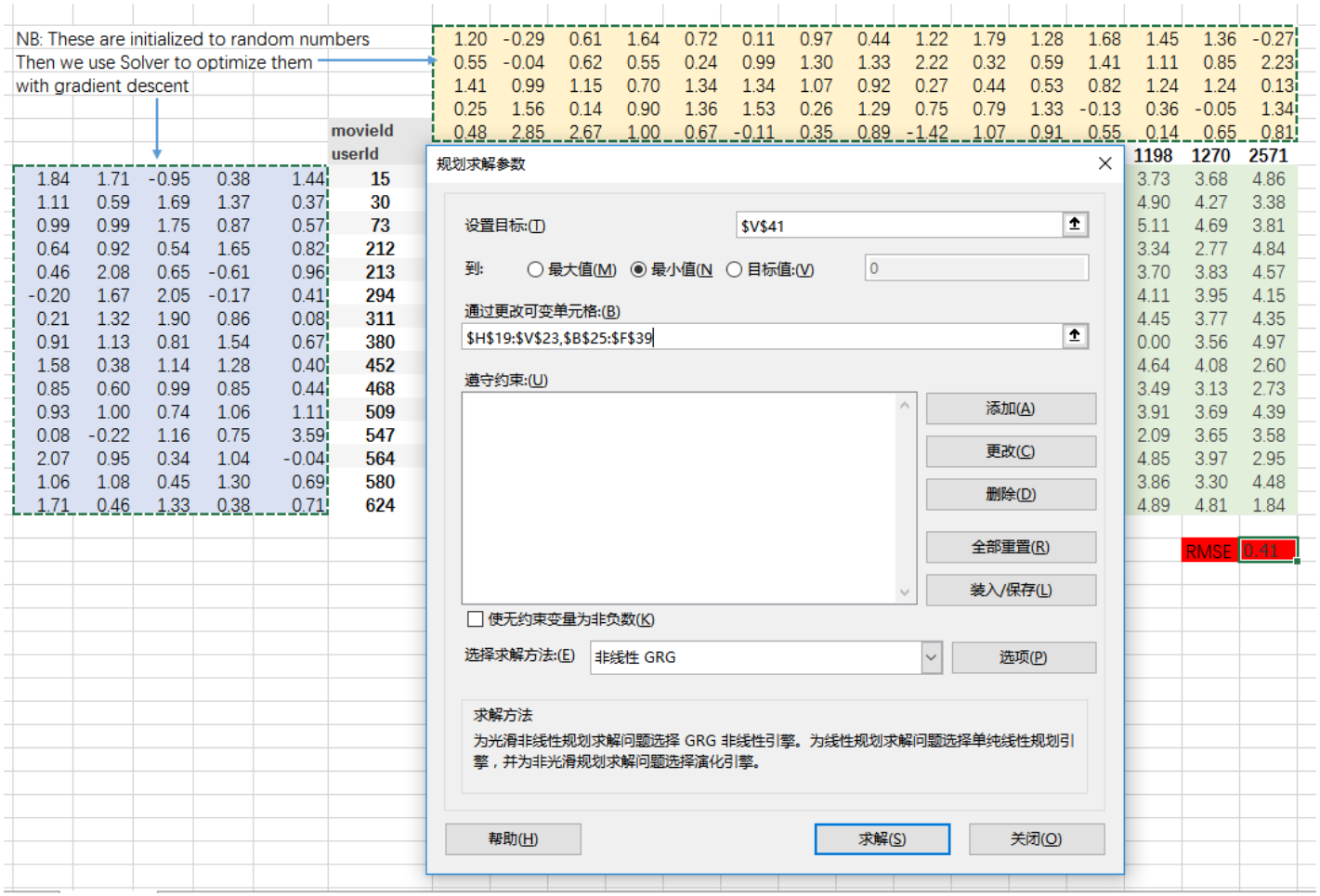
RMSE 2.82



$$rating_{pred} = \vec{u} \cdot \vec{m}$$

其中的向量$\vec{u}$ 和$\vec{m}$称为隐向量（latent vector），或称隐变量，其没有具体的意义，只是作为算法中，对应user以及movie一个潜在的信息，可以理解为——u[0]表示user对科幻类电影的喜好，m[0]是movie的科幻电影的“含量”。

利用Excel自带的<<规划求解>>的方法，以隐变量作为自变量，规划求解使得RMSE最小。



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

					movieId	1	110	260	296	318	356	480	527	589	593	608	1196	1198	1270	2571
						1.37	-0.92	0.11	1.56	0.79	0.08	1.18	0.35	1.75	1.70	1.23	1.74	1.55	1.57	-0.42
						0.56	-1.29	-0.97	0.81	0.46	0.99	1.17	1.61	3.49	0.73	0.94	1.33	0.93	0.83	2.47
						1.25	2.08	2.22	0.85	1.27	1.18	1.10	0.76	-0.61	0.71	0.80	0.77	1.17	1.10	-0.07
						0.32	1.96	0.48	0.82	1.43	1.71	0.05	1.51	0.31	0.48	0.98	-0.12	0.36	0.00	1.66
						0.25	7.81	8.05	0.52	0.14	-0.12	0.35	0.41	-6.00	0.60	0.48	0.56	0.26	0.41	0.53
2.51	1.52	-1.75	0.94	1.18	15	2.72	3.14	4.86	5.02	1.95	1.10	3.27	3.89	3.99	5.29	4.59	5.59	3.89	3.76	4.99
1.18	0.69	1.67	1.36	0.05	30	4.54	4.50	4.17	4.96	5.31	5.06	4.12	4.87	3.61	4.36	4.79	4.12	4.93	4.29	3.37
1.04	1.12	1.69	0.95	0.19	73	4.52	4.45	4.76	4.84	4.86	4.78	4.51	4.97	3.84	4.34	4.70	4.58	5.02	4.50	3.87
1.08	0.94	0.20	1.77	0.43	212	2.94	5.04	3.97	4.28	4.12	4.22	2.82	4.88	3.00	3.76	0.00	0.00	3.53	2.87	4.99
-0.28	2.56	1.63	-1.38	0.57	213	2.80	2.11	5.05	0.00	0.00	2.00	4.61	0.00	3.60	2.24	2.29	4.66	3.50	3.70	4.34
0.10	1.67	1.94	-0.05	0.20	294	3.54	3.24	4.27	0.00	3.26	3.83	4.29	4.21	3.62	0.00	0.00	4.01	4.02	3.76	3.96
-0.49	1.70	2.96	0.06	-0.16	311	3.95	3.24	3.56	3.09	4.20	5.24	4.62	4.84	4.26	2.43	3.33	3.58	4.25	3.82	4.19
1.17	1.15	0.63	1.58	0.37	380	3.65	4.76	4.17	4.78	4.56	4.63	3.63	0.00	3.93	4.25	4.75	4.06	0.00	3.64	5.11
1.67	0.46	1.10	1.27	0.13	452	4.38	3.69	3.86	5.03	4.77	4.05	3.83	4.15	3.44	4.64	4.68	4.29	4.80	4.28	2.52
1.02	0.64	0.80	0.95	0.19	468	3.12	3.25	3.26	3.67	3.50	3.27	2.95	0.00	0.00	3.33	3.52	3.24	3.51	3.10	2.78
1.51	0.94	0.13	1.51	0.61	509	3.41	5.37	5.16	4.77	4.02	3.70	3.30	4.66	2.65	4.42	4.60	4.13	4.07	3.55	4.47
-2.43	-2.44	3.63	1.12	13.06	547	3.51	0.00	0.00	5.00	5.00	2.01	2.97	5.00	0.00	5.00	4.98	2.51	2.00	3.52	3.50
1.96	0.89	0.39	0.94	0.19	564	4.04	1.19	2.20	4.98	0.00	3.09	3.90	3.93	5.45	4.82	4.58	4.89	4.72	4.33	3.00
1.43	1.02	0.10	1.47	0.46	580	3.25	4.04	3.78	4.57	3.88	3.70	3.22	4.62	3.69	4.21	4.44	3.99	3.93	3.39	4.57
2.03	0.48	0.79	0.84	0.37	624	4.41	0.00	4.84	5.10	0.00	2.95	4.00	0.00	2.82	4.98	4.58	4.89	4.92	4.62	1.85

RMSE 0.40

简单粗暴的Python方法

Collaborative filtering

In [*]:
```
val_idxs = get_cv_idxs(len(ratings))
wd=2e-4
n_factors = 50
```

In [52]:
```
cf = CollabFilterDataset.from_csv(path, 'ratings.csv', 'userId', 'movieId', 'rating')
learn = cf.get_learner(n_factors, val_idxs, 64, opt_fn=optim.Adam)
```

In [58]:
```
learn.fit(1e-2, 2, wds=wd, cycle_len=1, cycle_mult=2, use_wd_sched=True)
```

Epoch 100% 3/3 [00:19<00:00, 6.55s/it]

```
[ 0.       0.8272   0.82381]
[ 1.       0.67867  0.8697 ]
[ 2.       0.33393  0.82195]
```

Let's compare to some benchmarks. Here's some benchmarks on the same dataset for the popular Librec system for collaborative filtering. They show best results based on RMSE of 0.91. We'll need to take the square root of our loss, since we use plain MSE.

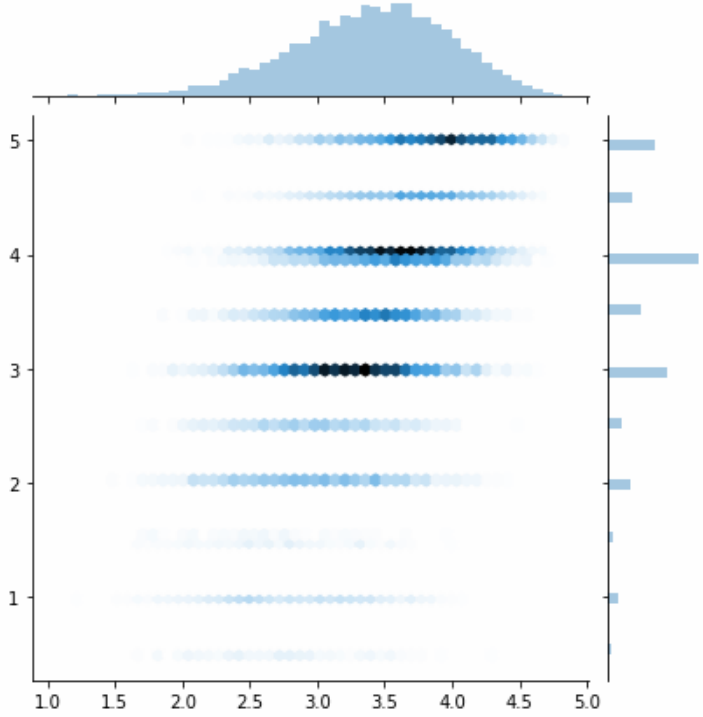
In [12]:
```
math.sqrt(0.82195)
```

Out[12]:
```
0.9068951062292475
```

简单粗暴的Python方法

In [13]:
```
preds = learn.predict()
```

In [14]:
```
y=learn.data.val_y
sns.jointplot(preds, y, kind='hex', stat_func=None);
```



点积

$$\begin{bmatrix} w & x \\ y & z \end{bmatrix} \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} wa & xb \\ yc & zd \end{bmatrix}$$

Dot product example

In [32]:
```
a = T([[1.,2,3],[3,4,6],[5,6,9]])
b = T([[2.,2,4],[10,10,11],[3,9,10]])
a,b
```

Out[32]:
```
(
 1  2  3
 3  4  6
 5  6  9
[torch.FloatTensor of size 3x3],
  2   2   4
 10  10  11
  3   9  10
[torch.FloatTensor of size 3x3])
```

In [40]:
```
a*b
```

Out[40]:
```
 2   4  12
30  40  66
15  54  90
[torch.FloatTensor of size 3x3]
```

Pytorch中，直接用T即可初始化一个Tensor，而点积运算直接可以通过*号进行

```
In [40]: a*b

Out[40]:
  2   4  12
 30  40  66
 15  54  90
[torch.FloatTensor of size 3x3]

In [59]: (a*b).sum(1)

Out[59]:
  18
 136
 159
[torch.FloatTensor of size 3]

In [60]: (a*b).sum(0)

Out[60]:
  47
  98
 168
[torch.FloatTensor of size 3]
```

使用sum()函数得到结果

Dot product model

```
u_uniq = ratings.userId.unique()
user2idx = {o:i for i,o in enumerate(u_uniq)}
ratings.userId = ratings.userId.apply(lambda x: user2idx[x])

m_uniq = ratings.movieId.unique()
movie2idx = {o:i for i,o in enumerate(m_uniq)}
ratings.movieId = ratings.movieId.apply(lambda x: movie2idx[x])

n_users=int(ratings.userId.nunique())
n_movies=int(ratings.movieId.nunique())
```

将userId与movieId重新编号，变成其在数组中的index，以便计算机读取，图右为再编号之后的movieId列表

24	24
25	25
26	26
27	27
28	28
29	29
	...
99974	473
99975	354
99976	355
99977	5577
99978	477
99979	478
99980	358
99981	479
99982	480
99983	359
99984	1225
99985	1240
99986	361
99987	126
99988	1260
99989	483
99990	362
99991	127
99992	364
99993	1299
99994	412
99995	486

In [80]:
```
class EmbeddingDot(nn.Module):
    def __init__(self, n_users, n_movies):
        super().__init__()
        self.u = nn.Embedding(n_users, n_factors)
        self.m = nn.Embedding(n_movies, n_factors)
        self.u.weight.data.uniform_(0,0.05)
        self.m.weight.data.uniform_(0,0.05)

    def forward(self, ratings, conts):
        users,movies = ratings[:,0],ratings[:,1]
        u,m = self.u(users),self.m(movies)
        return (u*m).sum(1)
```

In [81]:
```
x = ratings.drop(['rating', 'timestamp'],axis=1)
y = ratings['rating'].astype(np.float32)
```

In [82]:
```
data = ColumnarModelData.from_data_frame(path, val_idxs, x, y, ['userId', 'movieId'], 64)
```

In [83]:
```
wd=1e-5
model = EmbeddingDot(n_users, n_movies).cuda()
opt = optim.SGD(model.parameters(), 1e-1, weight_decay=wd, momentum=0.9)
```

In [84]:
```
fit(model, data, 3, opt, F.mse_loss)
```

Epoch 100% 3/3 [00:15<00:00, 5.09s/it]

```
[ 0.       1.66109  1.63607]
[ 1.       1.0887   1.29709]
[ 2.       0.93157  1.2139 ]
```

In [86]:
```
set_lrs(opt, 0.01)
```

In [87]:
```
fit(model, data, 3, opt, F.mse_loss)
```

Epoch 100% 3/3 [00:15<00:00, 5.01s/it]

```
[ 0.       0.46675  1.10683]
[ 1.       0.44249  1.09991]
[ 2.       0.41057  1.09848]
```

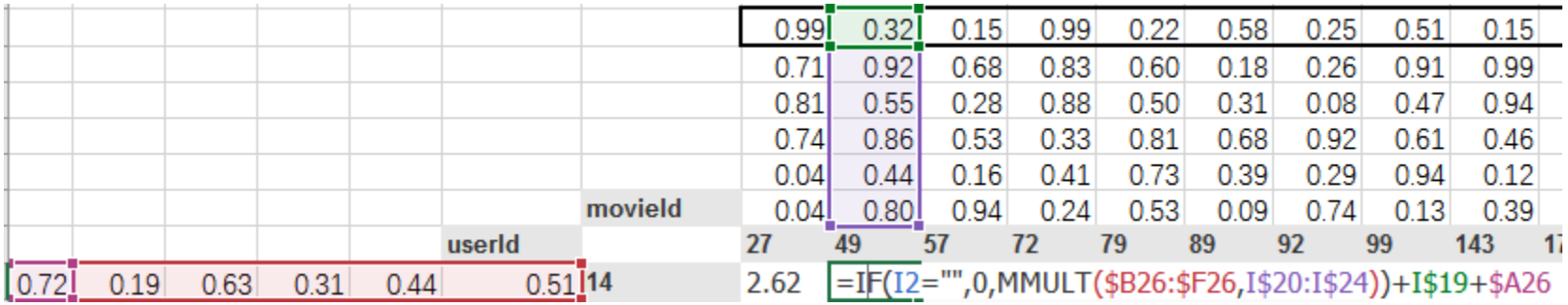
In [*]:
```
set_lrs(opt, 1e-3)
```

In [88]:
```
fit(model, data, 3, opt, F.mse_loss)
```

Epoch 100% 3/3 [00:15<00:00, 5.03s/it]

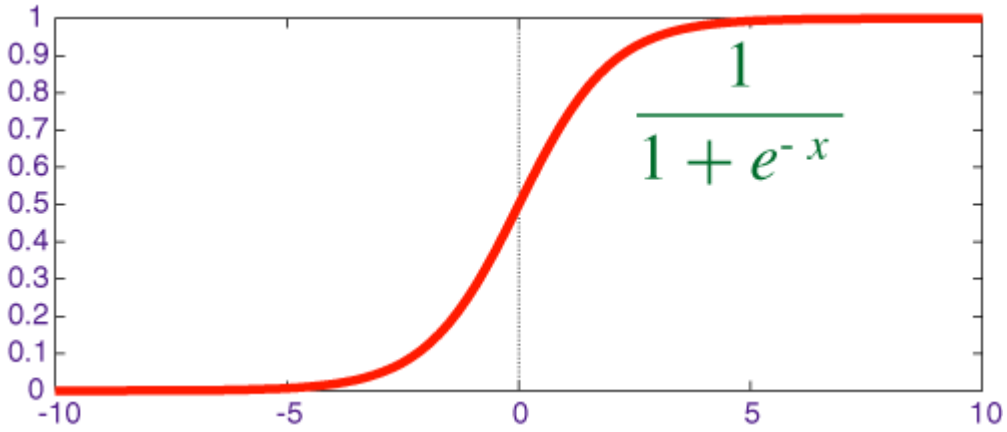
```
[ 0.       0.3956   1.09758]
[ 1.       0.40314  1.09764]
[ 2.       0.36906  1.09799]
```

带有偏置Bias的模型



								0.99	0.32	0.15	0.99	0.22	0.58	0.25	0.51	0.15
								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	17
0.72	0.19	0.63	0.31	0.44	0.51	14		2.62	=IF(I2="",0,MMULT($B26:$F26,I$20:I$24))+I$19+$A26								

为每部电影与每个user设定一个Bias的值，根据电影本身的人气度以及user的喜好，本身可以有一个bias的值，从而使得其他的属性更加容易判别。



利用sigmoid函数，将所有的预测值限定在评分范围内。

Bias

In [91]:
```
min_rating,max_rating = ratings.rating.min(),ratings.rating.max()
min_rating,max_rating
```

Out[91]:
```
(0.5, 5.0)
```

In [92]:
```
def get_emb(ni,nf):
    e = nn.Embedding(ni, nf)
    e.weight.data.uniform_(-0.01,0.01)
    return e

class EmbeddingDotBias(nn.Module):
    def __init__(self, n_users, n_movies):
        super().__init__()
        (self.u, self.m, self.ub, self.mb) = [get_emb(*o) for o in [
            (n_users, n_factors), (n_movies, n_factors), (n_users,1), (n_movies,1)
        ]]

    def forward(self, cats, conts):
        users,movies = cats[:,0],cats[:,1]
        um = (self.u(users)* self.m(movies)).sum(1)
        res = um + self.ub(users).squeeze() + self.mb(movies).squeeze()
        res = F.sigmoid(res) * (max_rating-min_rating) + min_rating
        return res
```

In [101]:
```
wd=2e-4
model = EmbeddingDotBias(cf.n_users, cf.n_items).cuda()
opt = optim.SGD(model.parameters(), 1e-1, weight_decay=wd, momentum=0.9)
```

In [102]:
```
fit(model, data, 3, opt, F.mse_loss)
```

Epoch 100% 3/3 [00:17<00:00, 5.98s/it]

```
[ 0.       0.8371   0.83869]
[ 1.       0.77577  0.81259]
[ 2.       0.80474  0.80821]
```

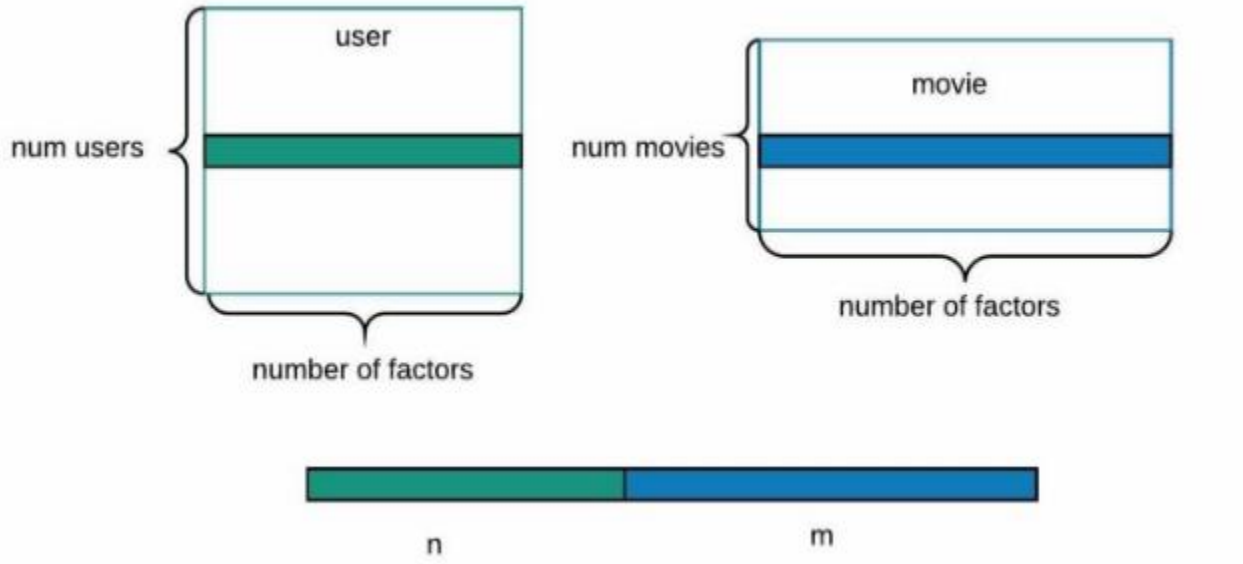
In [103]:
```
set_lrs(opt, 1e-2)
```

In [104]:
```
fit(model, data, 3, opt, F.mse_loss)
```

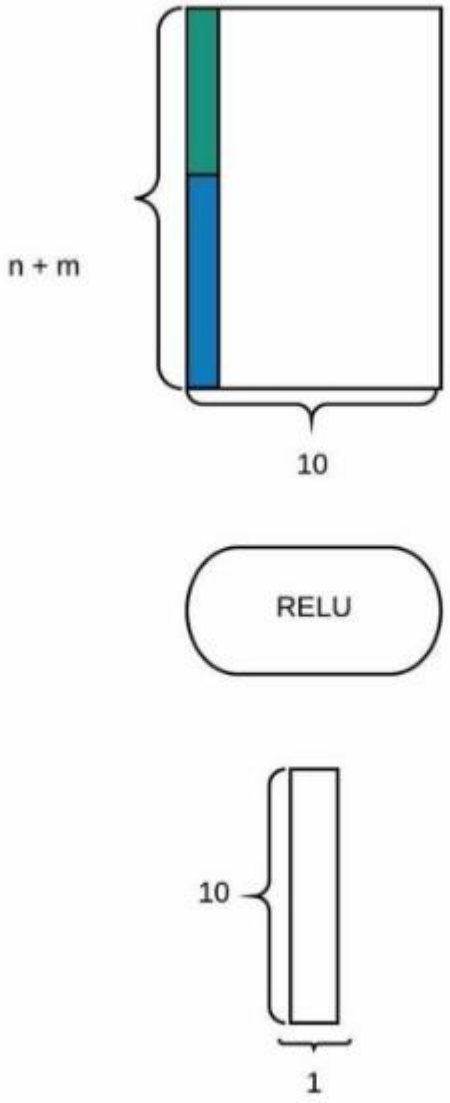
Epoch 100% 3/3 [00:18<00:00, 6.02s/it]

```
[ 0.       0.74764  0.8015 ]
[ 1.       0.73548  0.80027]
[ 2.       0.73524  0.80043]
```

利用神经网络实现



将用户和电影的输入值级联起来，得到一个n+m长度的vector作为输入



将合并的vector输入网络之中，通过两个全连接层以输出预测的分数。

Mini net

In [105]:
```
class EmbeddingNet(nn.Module):
    def __init__(self, n_users, n_movies, nh=10, p1=0.05, p2=0.5):
        super().__init__()
        (self.u, self.m) = [get_emb(*o) for o in [
            (n_users, n_factors), (n_movies, n_factors)]]
        self.lin1 = nn.Linear(n_factors*2, nh)
        self.lin2 = nn.Linear(nh, 1)
        self.drop1 = nn.Dropout(p1)
        self.drop2 = nn.Dropout(p2)

    def forward(self, cats, conts):
        users,movies = cats[:,0],cats[:,1]
        x = self.drop1(torch.cat([self.u(users),self.m(movies)], dim=1))
        x = self.drop2(F.relu(self.lin1(x)))
        return F.sigmoid(self.lin2(x)) * (max_rating-min_rating+1) + min_rating-0.5
```

In [106]:
```
wd=1e-5
model = EmbeddingNet(n_users, n_movies).cuda()
opt = optim.Adam(model.parameters(), 1e-3, weight_decay=wd)
```

In [108]:
```
fit(model, data, 3, opt, F.mse_loss)
```

Epoch 100% 3/3 [00:20<00:00, 6.89s/it]

```
[ 0.      0.80272  0.78703]
[ 1.      0.78137  0.78761]
[ 2.      0.76988  0.78973]
```

In [111]:
```
set_lrs(opt, 1e-4)
```

In [112]:
```
fit(model, data, 3, opt, F.mse_loss)
```

Epoch 100% 3/3 [00:21<00:00, 7.03s/it]

```
[ 0.      0.68928  0.79175]
[ 1.      0.70755  0.79171]
[ 2.      0.70979  0.79076]
```

目录 Contents

weights:		
b	const	30
a	slope	2
	x	y=a*x + b
	26	82
	97	224
	80	190
	98	226
	50	130
	43	116
	51	132
	7	44
	52	134
	59	148
	78	186
	89	208
	14	58
	44	118
	19	68
	48	126
	69	168
	99	228
	88	206
	10	50
	36	102
	28	86
	97	224
	42	114

利用一个一维线性模型进行实验，首先随机生成一些x，通过一次函数得到y值，将这些数据组作为实验的数据。

误差表示为

$$err = (ax + b - y)^2$$

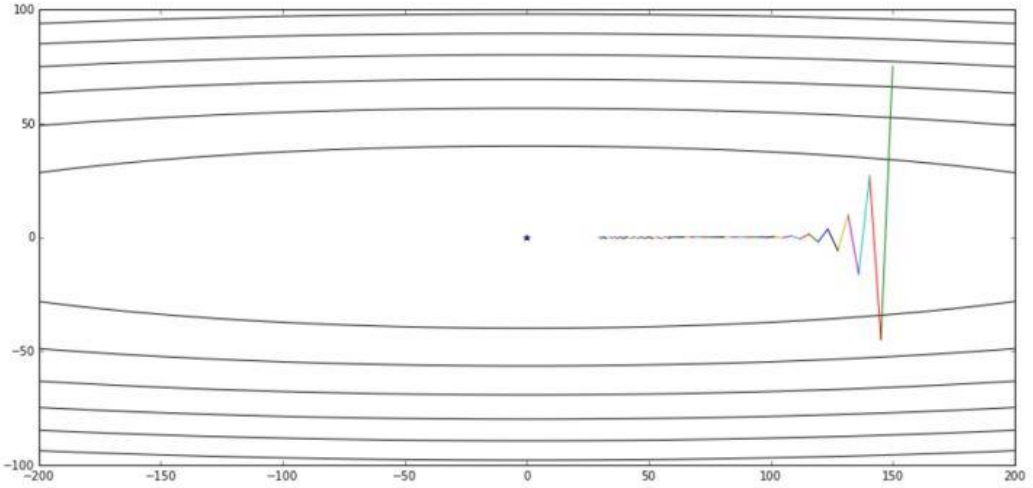
我们分别对a和b求导

$$\frac{de}{db} = 2(ax + b - y) \quad \frac{de}{da} = 2x(ax + b - y)$$

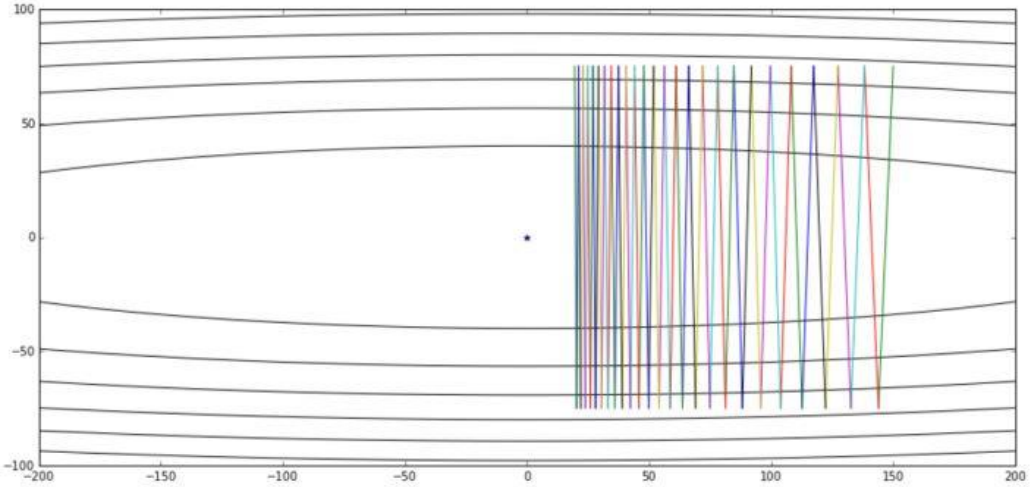
令

$$a \leftarrow a - \frac{de}{da} \cdot lr$$

b同理，更新的a, b用于进行下一个数据的计算


100
50
0
−50
−100
−200
−150
−100
−50
0
50
100
150
200

学习率较小


100
50
0
−50
−100
−200
−150
−100
−50
0
50
100
150
200

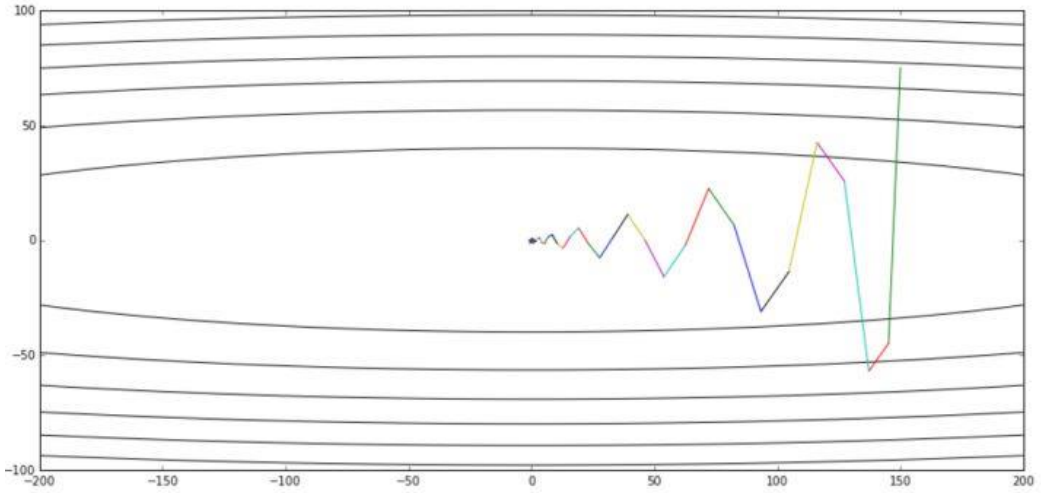
学习率偏大

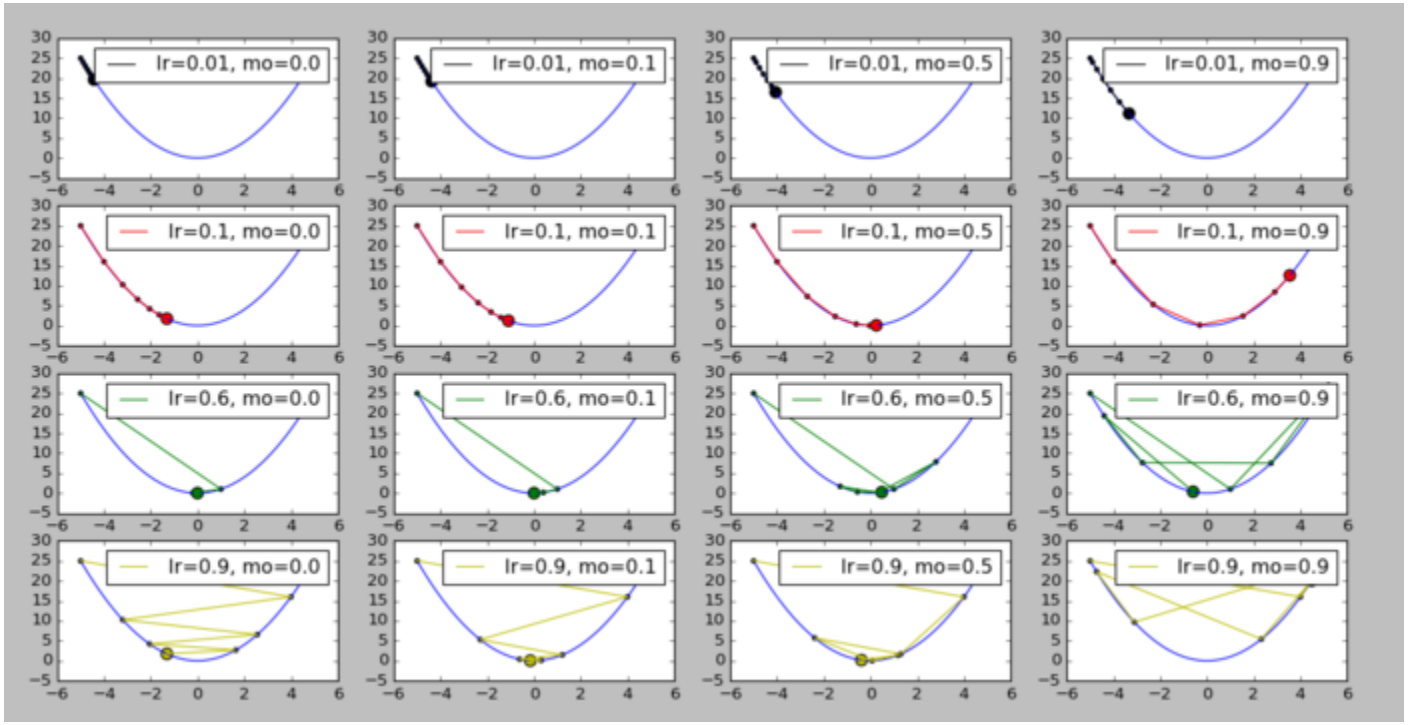
动量 momentum

$$v_i = \frac{de}{da} \cdot (1 - momentum) + v_{i-1} \cdot momentum$$

$$a \leftarrow a - v_i \cdot lr$$

- 当本次梯度下降的方向与上次更新量v的方向**相同**时，上次的更新值能够对本次的更新起到一个正向**加速**的作用。
- 当本次梯度下降的方向与上次更新量v的方向**相反**时，上次的更新值能够对本次的更新起到一个**减速**的作用。





在学习率较小的时候，适当的momentum能够起到一个加快收敛速度的作用。
在学习率较大的时候，适当的momentum能够起到一个减小收敛时震荡幅度的作用。

RMSProp (root mean square prop)

RMSProp算法给每一个权值一个变量MeanSquare(w,t)用来记录第t次更新步长时前t-1次的累积平方梯度与该次更新梯度平方的内插值，然后再用第t次的梯度除以该内插值的开根值，得到学习步长的更新比例，根据此比例去得到新的更新步长。

$$S_a = \beta S_a + (1-\beta)(da)^2$$

$$a = a - lr \cdot \frac{da}{\sqrt{S_a + \epsilon}}$$

其中，ϵ用于防止分母为零，一般取10^{-8}

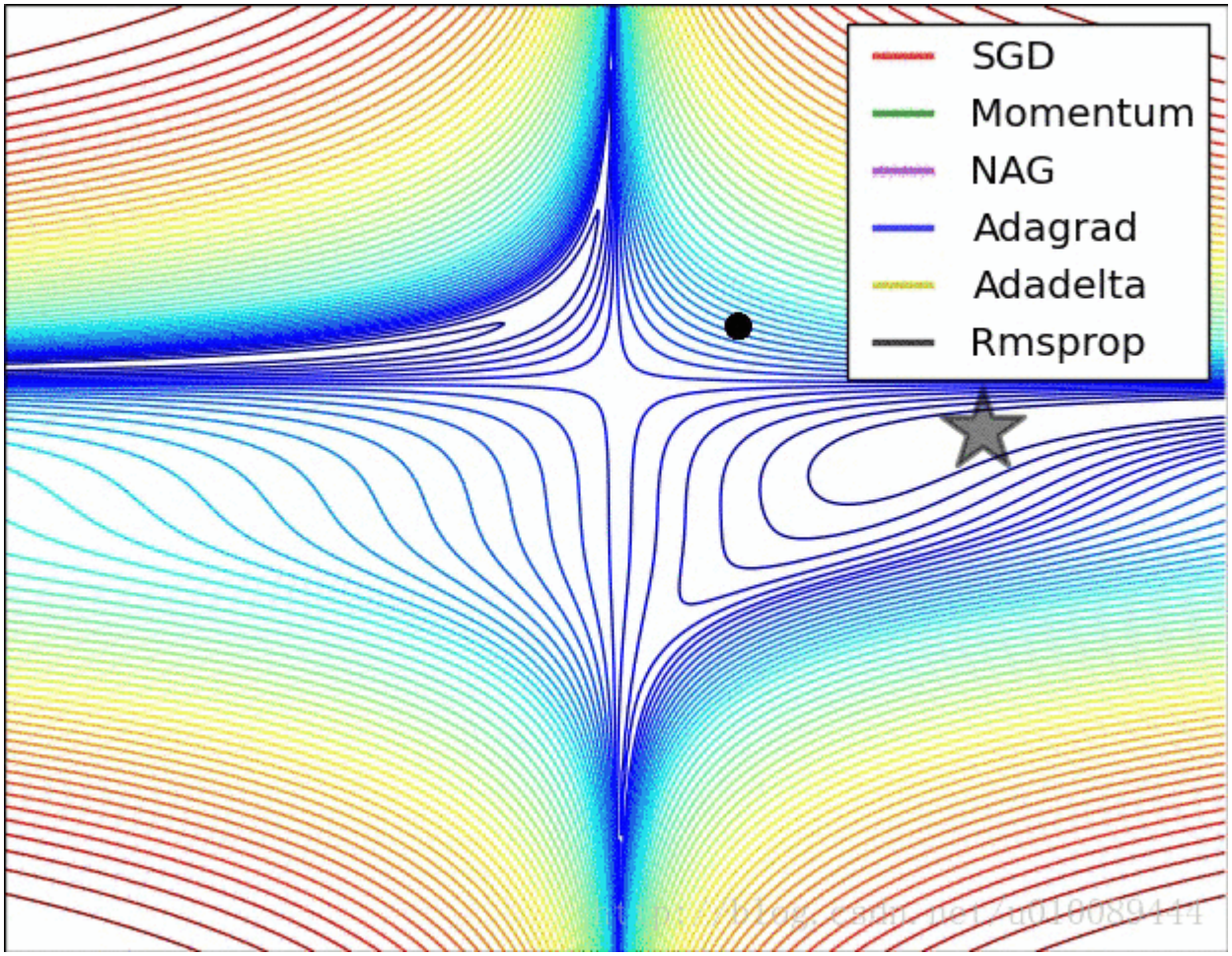
Adam(Adaptive Moment Estimation)

Adam(Adaptive Moment Estimation)是另一种自适应学习率的方法。它利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率,也就是RMSProp与momentum的结合。
Adam的优点主要在于经过偏置校正后，每一次迭代的学习率都有个确定范围，使得参数比较平稳。公式如下：

$$v = da \cdot (1 - m) + v \cdot m \qquad S_a = \beta S_a + (1 - \beta)(da)^2$$

$$a = a - lr \cdot \frac{v}{\sqrt{S_a + \epsilon}}$$

由于要除以较大的S，因而初始设定的学习率要偏大


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop

谢 谢！