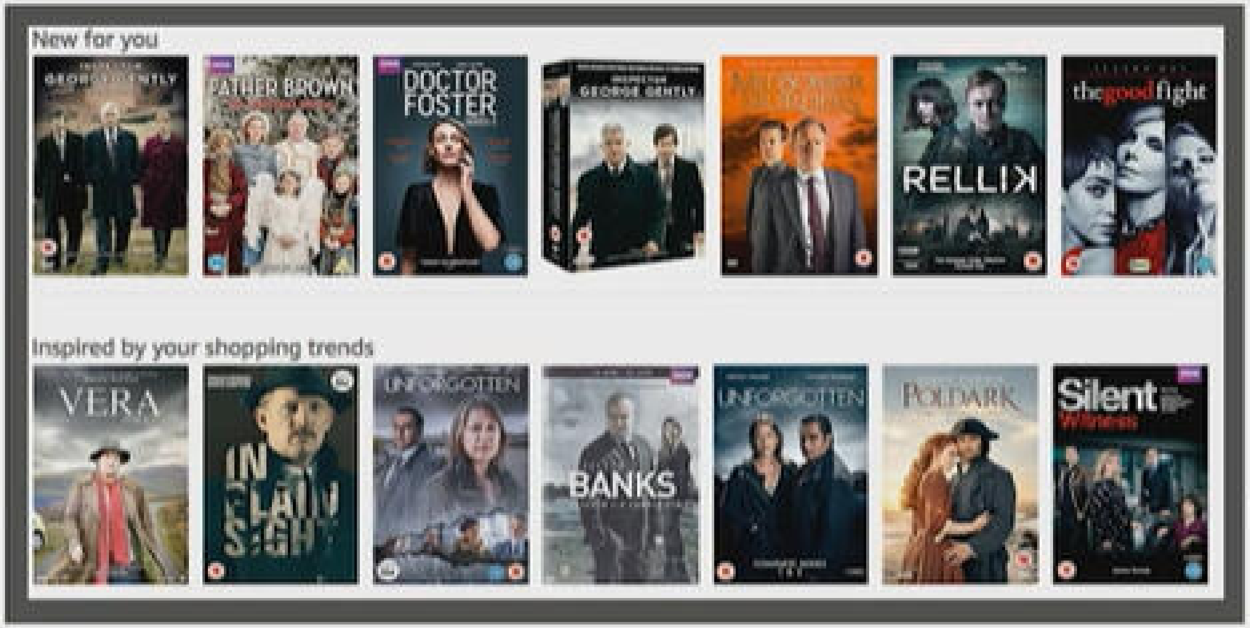
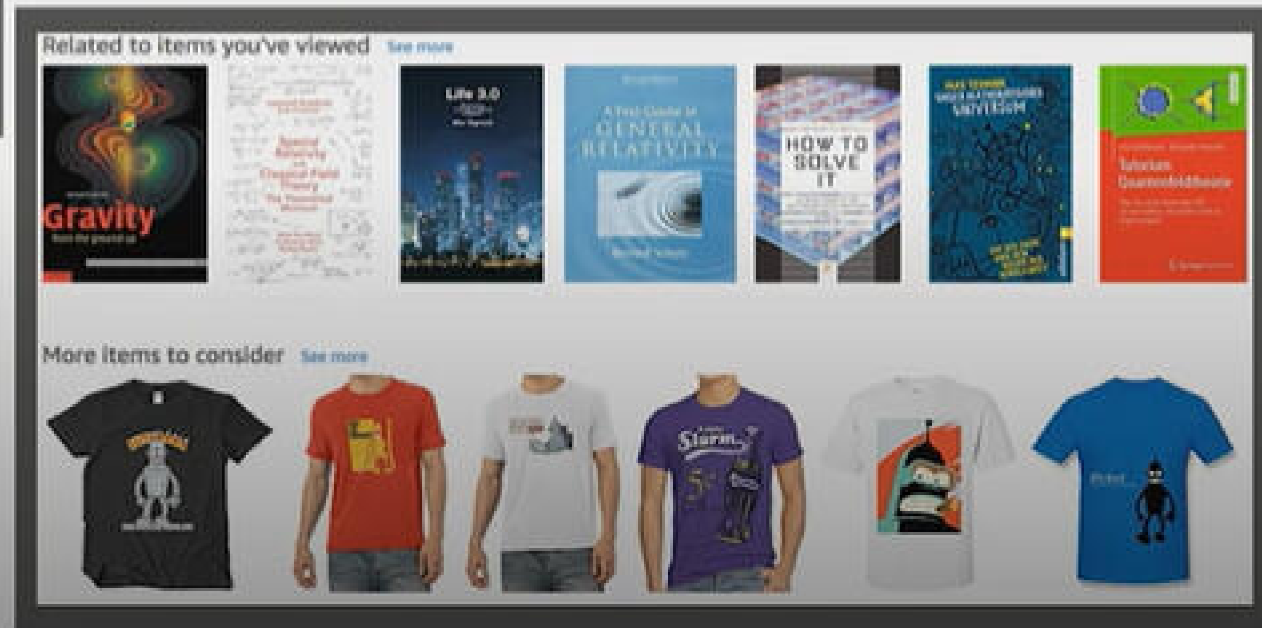
Motivation Contd.

aws training and certification



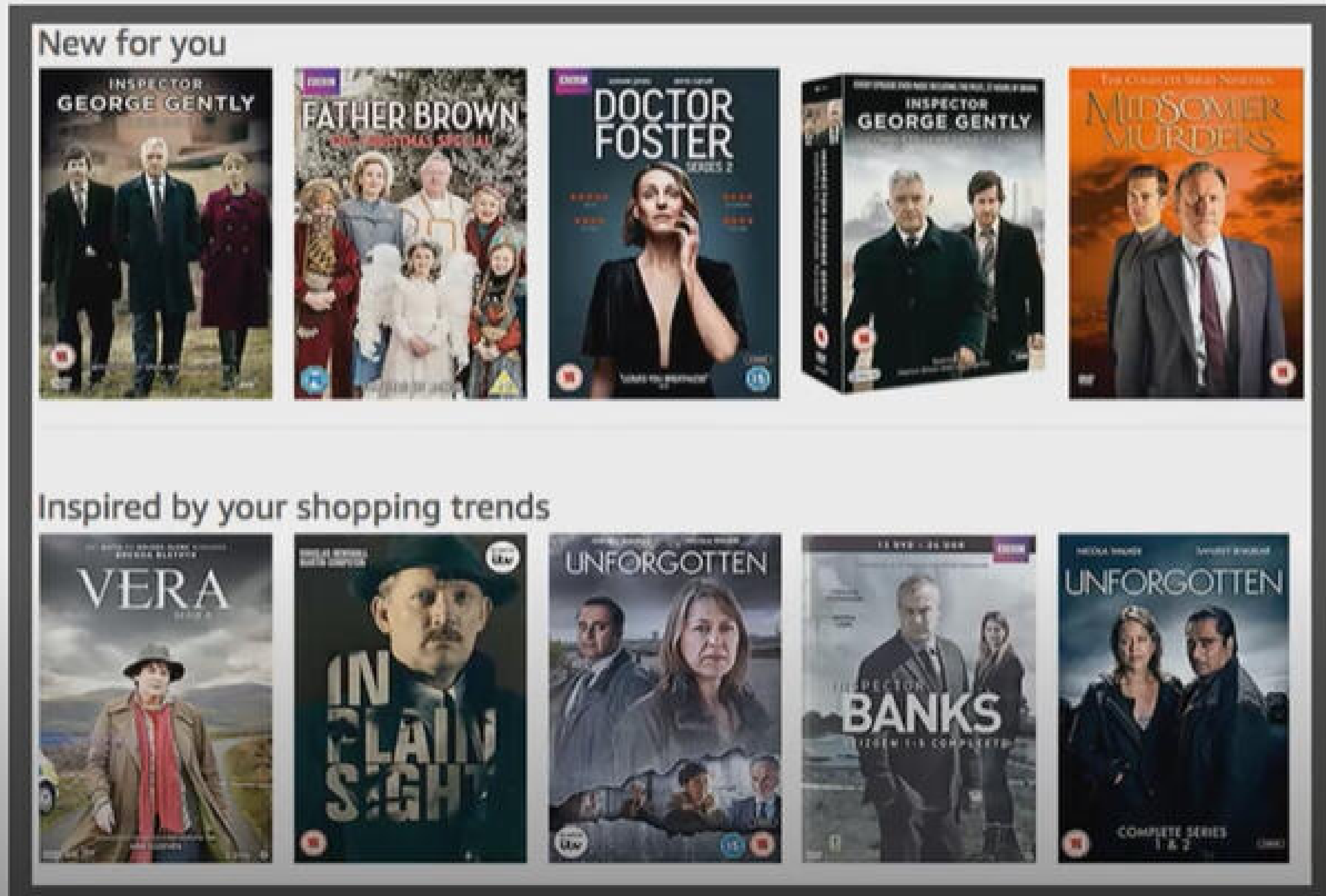
My Profile – amazon.co.uk



02:16 / 54:31

Collaborative Filtering Contd.

aws training and certification

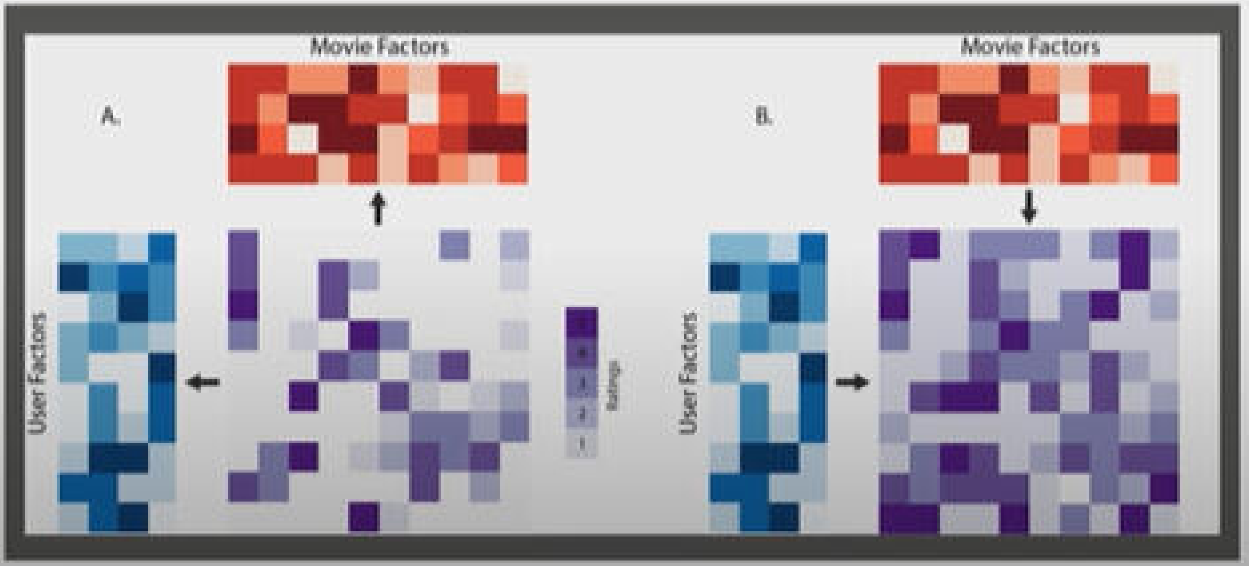


05:04 / 54:31

Matrix Factorization

aws training and certification

- **Matrix Factorization** factorizes a matrix to separate matrices, that when multiplied approximate to the completed matrix.
- For the sake of efficiency we would like to factorize the matrix to a long and a wide matrices.



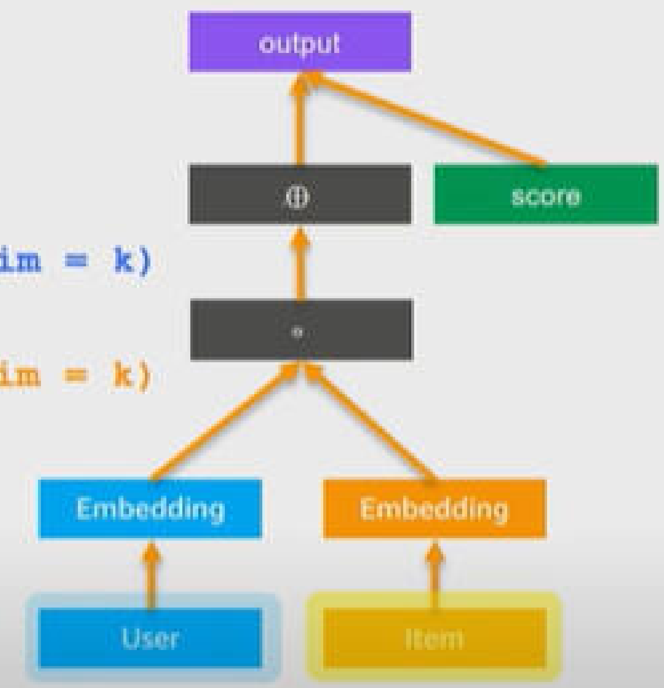
https://www.oreilly.com/ideas/deep-matrix-factorization-using-apache-mxnet?cmp=tw-data-na-article-engagement_sponsored+kibird

05:12 / 54:31

Linear Model

aws training and certification

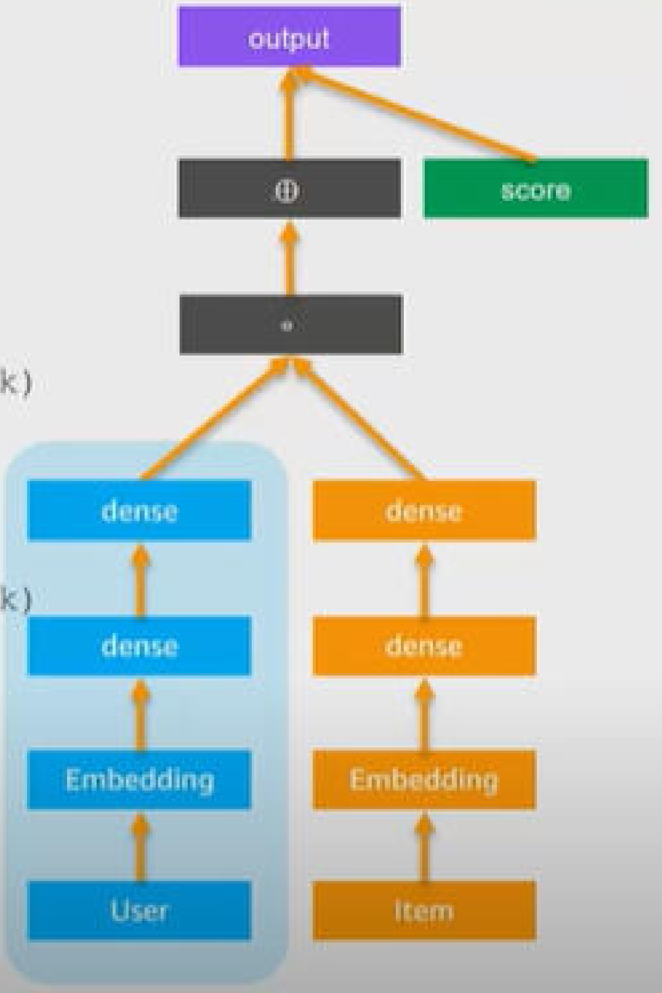
```
def plain_net(k):
# input
   user = mx.symbol.Variable('user')
   item = mx.symbol.Variable('item')
   score = mx.symbol.Variable('score')
# user feature lookup
   user = mx.symbol.Embedding(data = user, input_dim = max_user, output_dim = k)
# item feature lookup
   item = mx.symbol.Embedding(data = item, input_dim = max_item, output_dim = k)
# predict by the inner product, which is elementwise product and then sum
   pred = user * item
   pred = mx.symbol.sum(data = pred, axis = 1)
   pred = mx.symbol.Flatten(data = pred)
# loss layer
   pred = mx.symbol.LinearRegressionOutput(data = pred, label = score)
return pred

net1 = plain_net(64)
mx.viz.plot_network(net1)
```



Adding Non-Linearity

aws training and certification

```
def get_one_layer_mlp(hidden, k):
# input
    user = mx.symbol.Variable('user')
    item = mx.symbol.Variable('item')
    score = mx.symbol.Variable('score')
# user latent features
    user = mx.symbol.Embedding(data = user, input_dim = max_user, output_dim = k)
    user = mx.symbol.Activation(data = user, act_type='relu')
    user = mx.symbol.FullyConnected(data = user, num_hidden = hidden)
# item latent features
    item = mx.symbol.Embedding(data = item, input_dim = max_item, output_dim = k)
# predict by the inner product
    pred = user * item
    pred = mx.symbol.sum(data = pred, axis = 1)
    pred = mx.symbol.Flatten(data = pred)
# loss layer
    pred = mx.symbol.LinearRegressionOutput(data = pred, label = score)
    return pred
```

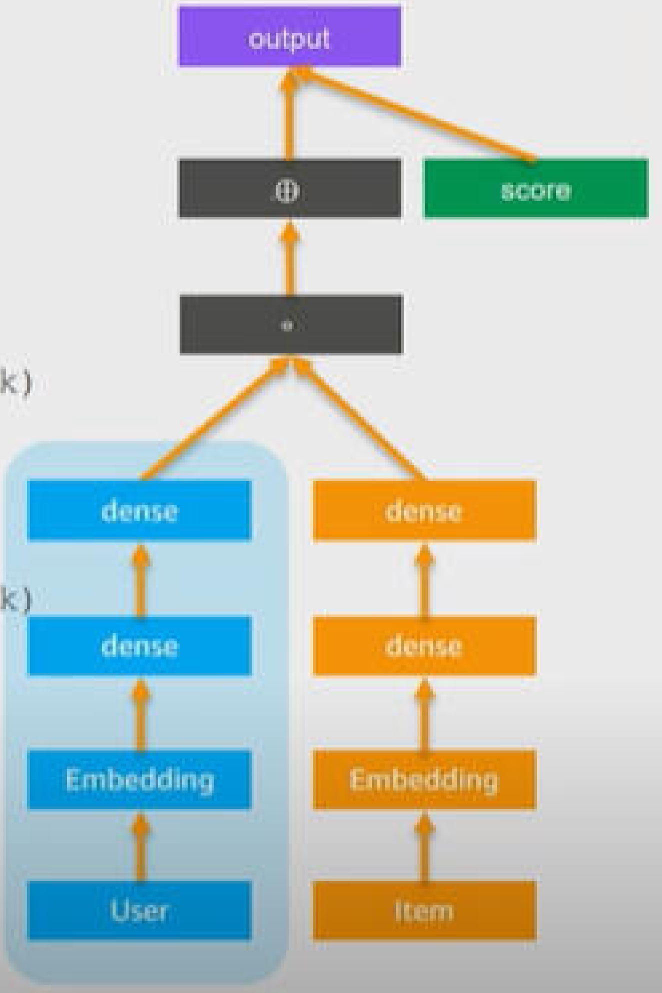


07:40 / 54:31

Adding Non-Linearity

aws training and certification

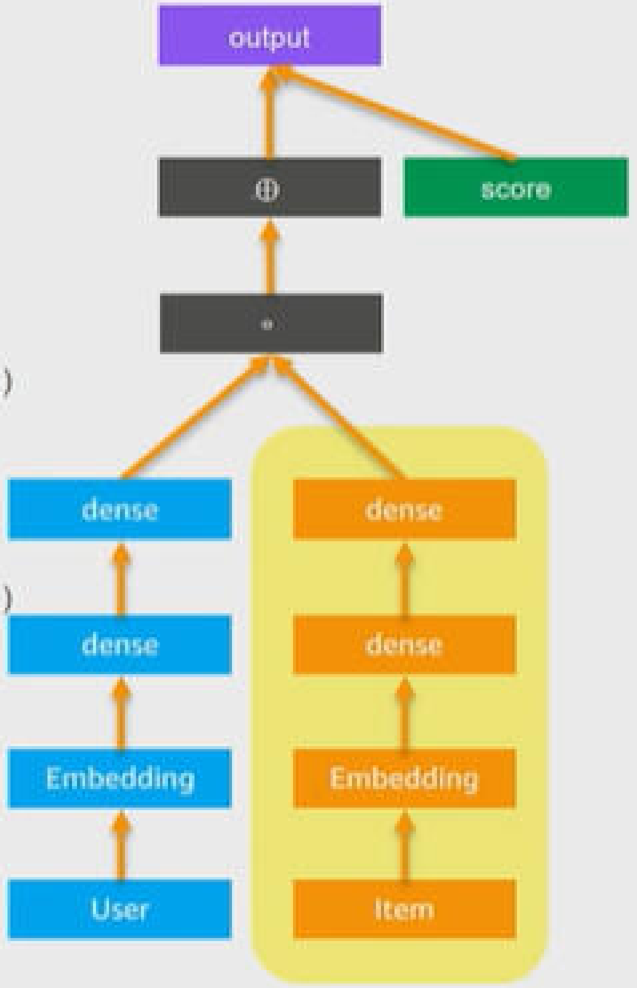
```
def get_one_layer_mlp(hidden, k):
# input
    user = mx.symbol.Variable('user')
    item = mx.symbol.Variable('item')
    score = mx.symbol.Variable('score')
# user latent features
    user = mx.symbol.Embedding(data = user, input_dim = max_user, output_dim = k)
    user = mx.symbol.Activation(data = user, act_type='relu')
    user = mx.symbol.FullyConnected(data = user, num_hidden = hidden)
# item latent features
    item = mx.symbol.Embedding(data = item, input_dim = max_item, output_dim = k)
# predict by the inner product
    pred = user * item
    pred = mx.symbol.sum(data = pred, axis = 1)
    pred = mx.symbol.Flatten(data = pred)
# loss layer
    pred = mx.symbol.LinearRegressionOutput(data = pred, label = score)
    return pred
```



07:40 / 54:31

Adding Non-Linearity Contd.

```
def get_one_layer_mlp(hidden, k):
# input
   user = mx.symbol.Variable('user')
   item = mx.symbol.Variable('item')
   score = mx.symbol.Variable('score')
# user latent features
   user = mx.symbol.Embedding(data = user, input_dim = max_user, output_dim = k)
   user = mx.symbol.Activation(data = user, act_type='relu')
   user = mx.symbol.FullyConnected(data = user, num_hidden = hidden)
# item latent features
   item = mx.symbol.Embedding(data = item, input_dim = max_item, output_dim = k)
   item = mx.symbol.Activation(data = item, act_type='relu')
   item = mx.symbol.FullyConnected(data = item, num_hidden = hidden)
# predict by the inner product
   pred = user * item
   pred = mx.symbol.sum(data = pred, axis = 1)
   pred = mx.symbol.Flatten(data = pred)
# loss layer
   pred = mx.symbol.LinearRegressionOutput(data = pred, label = score)
   return pred
```



Embedding Layer

- An Embedding Layer is where a network extracts the importance of features from data.
- Embedding is frequently used in NLP. For instance in sentiment analysis, embedding distills sentiment information from words.

https://www.oreilly.com/ideas/deep-matrix-factorization-using-apache-mxnet?cmp=tw-data-na-article-engagement_sponsored+kibird

Loading Data into an Array

```
train_data_iter = gluon.data.DataLoader(SparseMatrixDataset(train_data, train_label),
                                                      shuffle=True, batch_size=batch_size)

test_data_iter = gluon.data.DataLoader(SparseMatrixDataset(test_data, test_label),
                                                      shuffle=True, batch_size=batch_size)

class SparseMatrixDataset(gluon.data.Dataset):
      def __init__(self, data, label):
            assert data.shape[0] == len(label)
            self.data = data
            self.label = label
            if isinstance(label, ndarray.NDArray) and len(label.shape) == 1:
                   self._label = label.asnumpy()
            else:
                  self._label = label

      def __getitem__(self, idx):
            return self.data[idx, 0], self.data[idx, 1], self.label[idx]

      def __len__(self):
            return self.data.shape[0]
```

Defining the Network

aws training and certification

```
class MFBlock(gluon.Block):
    def __init__(self, max_users, max_items, num_emb, dropout_p=0.5):
        super(MFBlock, self).__init__()
        self.max_users = max_users
        self.max_items = max_items
        self.dropout_p = dropout_p
        self.num_emb = num_emb
        with self.name_scope():
            self.user_embeddings = gluon.nn.Embedding(max_users, num_emb)
            self.item_embeddings = gluon.nn.Embedding(max_items, num_emb)

    def forward(self, users, items):
        a = self.user_embeddings(users)
        b = self.item_embeddings(items)
        predictions = a * b
        predictions = nd.sum(predictions, axis=1)
        return predictions
```


12:51 / 54:31

Choosing the Optimizer

```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
{'learning_rate': lr, 'wd': wd, 'momentum': 0.9})
```

Training the Model

aws training and certification

```
epochs = 10
def train(data_iter, net):
    for e in range(epochs):
        print("epoch: {}".format(e))
        for i, (user, item, label) in enumerate(train_data_iter):
            user = user.as_in_context(ctx).reshape((batch_size,))
            item = item.as_in_context(ctx).reshape((batch_size,))
            label = label.as_in_context(ctx).reshape((batch_size,))
            with mx.autograd.record():
                output = net(user, item)
                loss = loss_function(output, label)
                loss.backward()
                net.collect_params().values()
            trainer.step(batch_size)
        print("EPOCH {}: RMSE ON TRAINING and TEST: {}. {}".format(e,
                                                eval_net(train_data_iter, net)),
                                                    eval_net(test_data_iter, net)))
return output
```

Adding Non-Linearity

```
class MFBlock(gluon.Block):
    def __init__(self, max_users, max_items, num_emb, dropout_p=0.5):
        super(MFBlock, self).__init__()
        self.max_users = max_users
        self.max_items = max_items
        self.dropout_p = dropout_p
        self.num_emb = num_emb
        with self.name_scope():
            self.user_embeddings = gluon.nn.Embedding(max_users, num_emb)
            self.item_embeddings = gluon.nn.Embedding(max_items, num_emb)
            self.dense = gluon.nn.Dense(num_emb, activation='relu')

    def forward(self, users, items):
        a = self.user_embeddings(users)
        a = self.dense(a)

        b = self.item_embeddings(items)
        b = self.dense(b)

        predictions = a * b
        predictions = nd.sum(predictions, axis=1)
        return predictions
```

Limitations

- Matrix Factorization is ideal for small catalogues and can perform based on small amounts of data.
- As the catalogues get larger, memory becomes a challenge.

Limitations Contd.

aws training and certification

- For instance MovieLens 20M has 27278 items and 138493 users. User x Item matrix will have a dimension of 138,493 X 27,278 = 3,777,812,054. From all possible ratings the dataset includes only 20000263. This means only 0.05 percent of the dataset contains data and 99.95 percent is just sparsity.

Storing the Matrix

<u>**Dense**</u>	<u>**Sparse**</u>
3.7B entries	20M non-zero entries
Each entry: •Rating: 1 byte	Each entry: •Rating: 1 byte •Movie_id: 32-bit integer •User_id: 32-bit integer
3.7 GB	**180 MB**

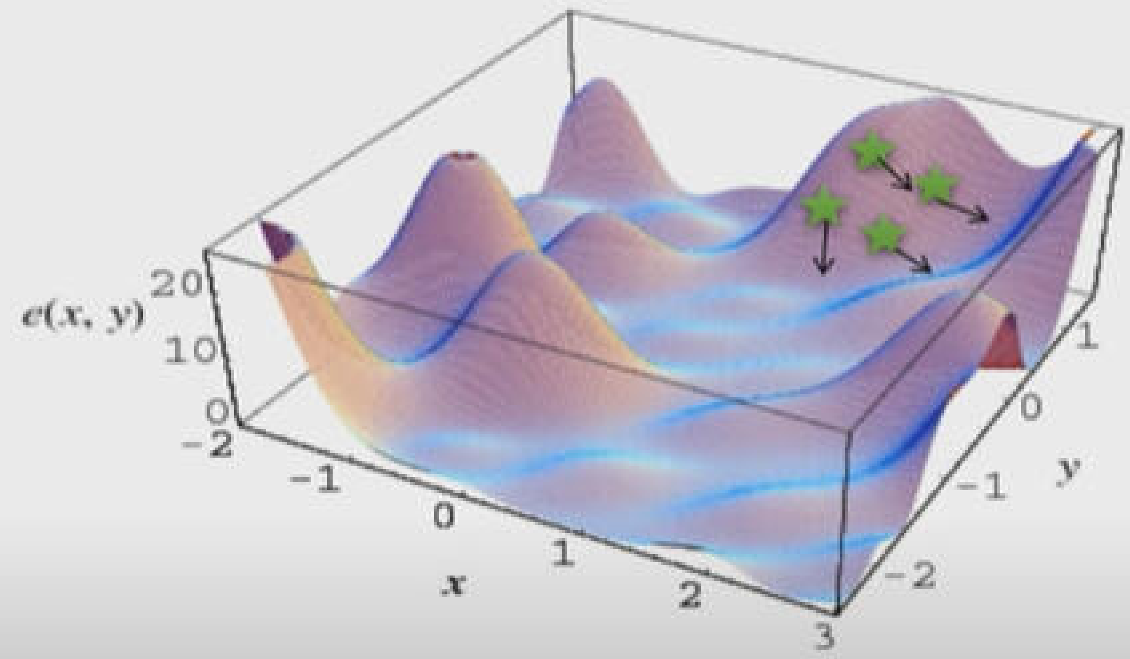
Sparse is 20x smaller!

ref: Leo Dirac – re:Invent 2016

The Scaling Issue Contd.

aws training and certification

- DiFacto or Distributed Factorization Machines are capable of distributing computation of sub-gradients on mini-batches asynchronously and can thus distribute load to several machines.



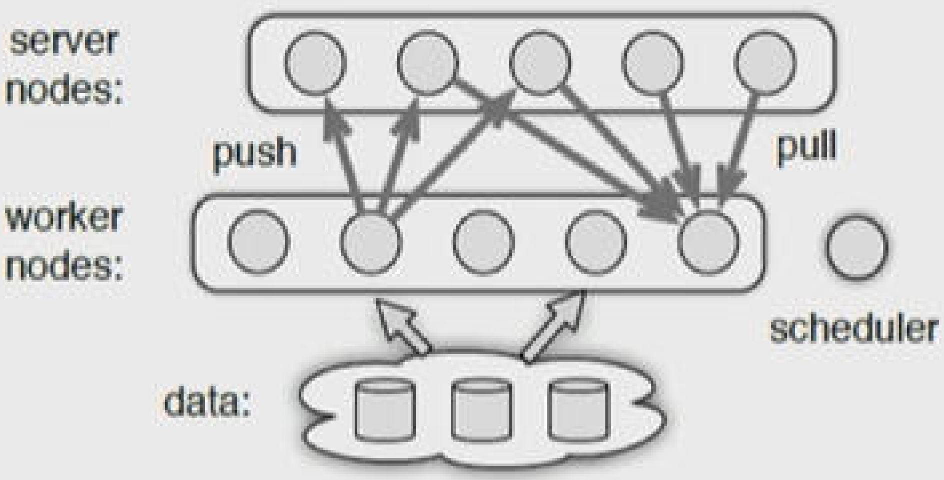
asynch SGD

20:08 / 54:31

The Scaling Issue

aws training and certification

- Factorization Machines take advantage of combining Support Vector Machines and MF in order to scale and deal with sparsity.
- The problem is that FM runs on a single machine and has huge memory requirements.



parameter server

(Alex Smola et al. - 2016)

asynch SGD

19:22 / 54:31

Content (Feature)-Based

aws training and certification

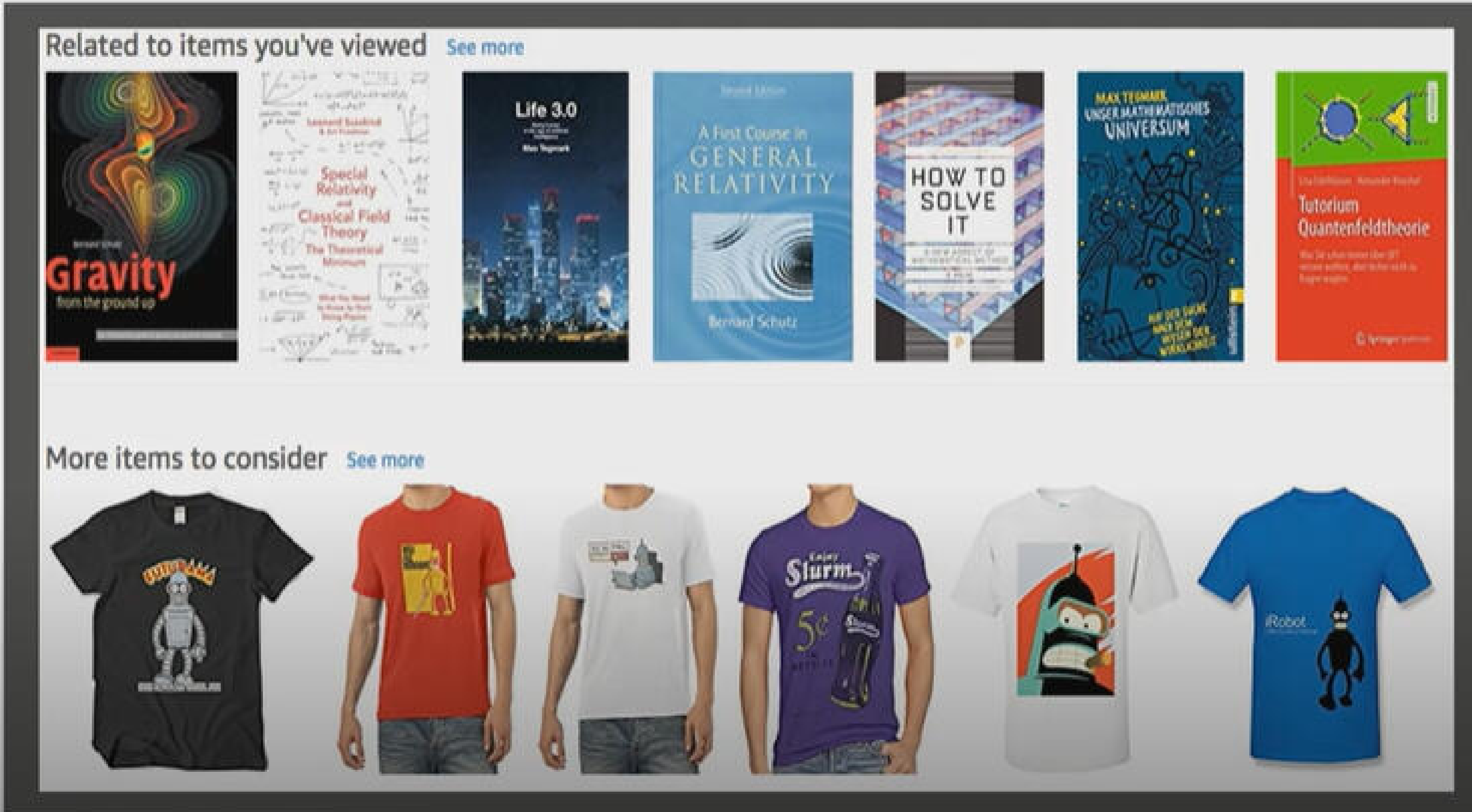
- We might not have user data, but there is often a wealth of product information available. We can use this data in order to recommend similar products to a user.

code	category	sub category	weight	price	colour	dimentions
itm1	1	1	20	123.1	blue	23x12x19
itm2	2	1	20	900	white	23x12x20
itm3	1	1	22	123.1	green	20x10x18
itm4	3	1	20	600	blue	23x12x22
itm5	1	2	19	200	yellow	23x12x23
itm6	1	1	1	12	red	2x1x3
itm7	1	1	900	2000	blue	100x80x99
itm8	9	8	20	6000	grey	99x99x99
itm9	7	5	1000	123.1	blue	123x5x8
itm10	9	8	20	5000	brown	99x99x99

21:08 / 54:31

Content Based

aws training and certification

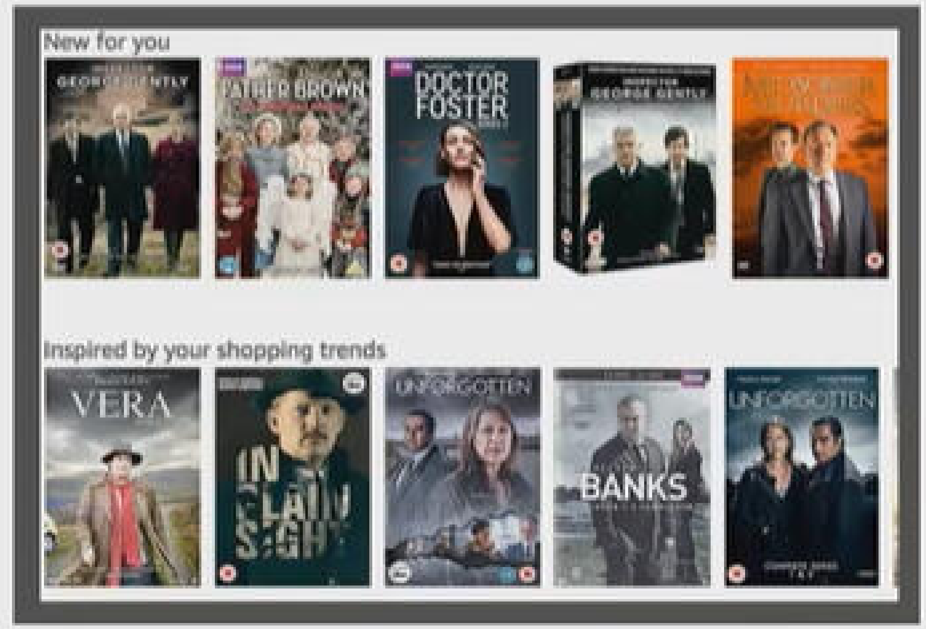


22:13 / 54:31

Untapped Data

- There is still a wealth of information we have not tapped into.
  - Movies have images.
  - Images can be captioned.
  - Products have names, while we have so far reduced them to item numbers.
  - TV series have episode names.
  - Products have verbose reviews.
- We can harvest all of this information and enrich our recommendation system.

There is a strong similarity in the ambience and composition of these images:

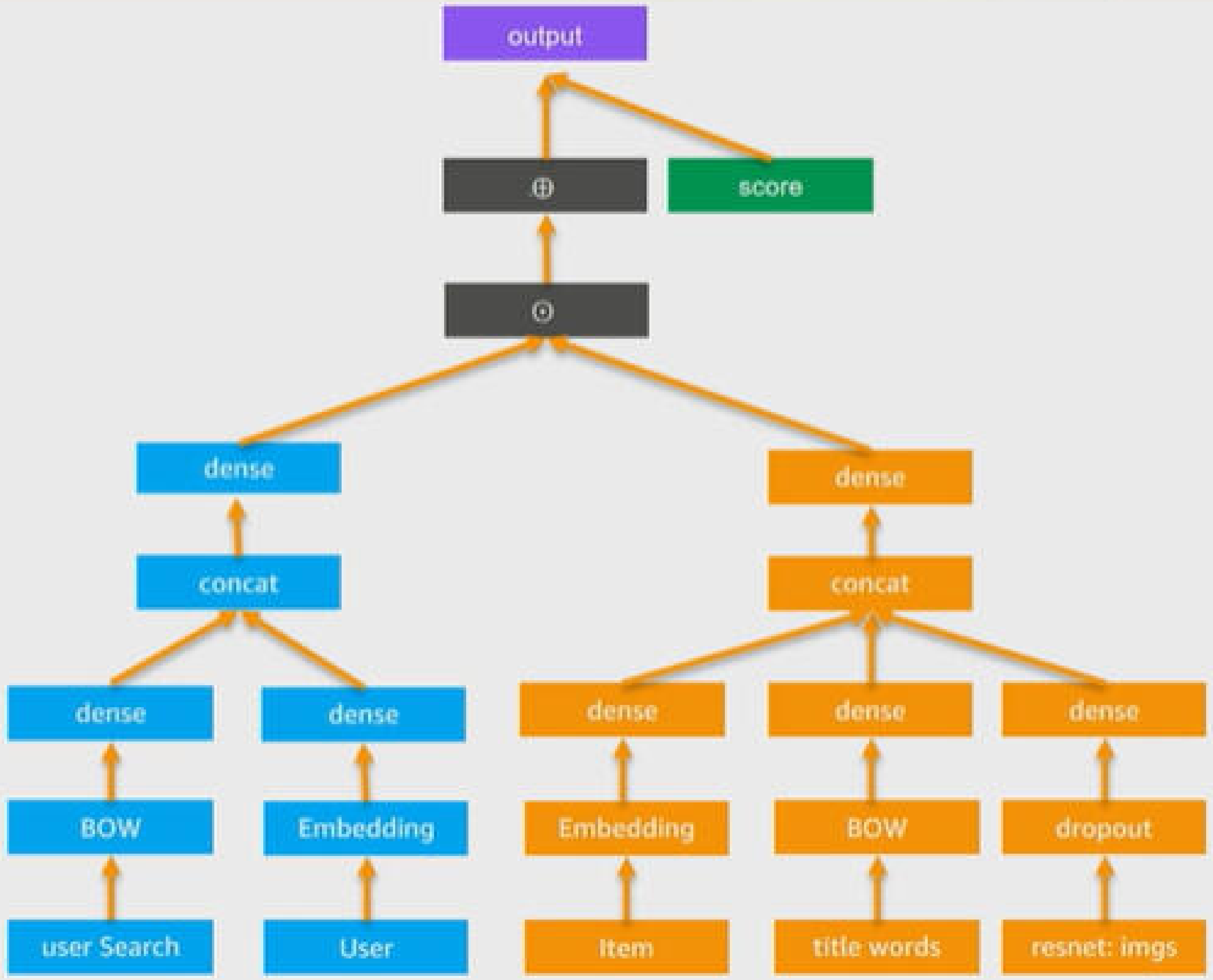


Deep Structure Semantic Models

aws training and certification

- A Matrix Factorization solution in its core is multiplication of two matrices.
- Neural Networks are good at picking up semantic intent at phrase/sentence level.
- Neural Networks are great at image captioning.
- The output of a network is a tensor.
- So we can use the output of several networks as our embedding layer for an enriched recommendation system.

DSSM Contd.


output
⊕
score
⊙
dense
dense
concat
concat
dense
dense
dense
dense
dense
BOW
Embedding
Embedding
BOW
dropout
user Search
User
Item
title words
resnet: imgs

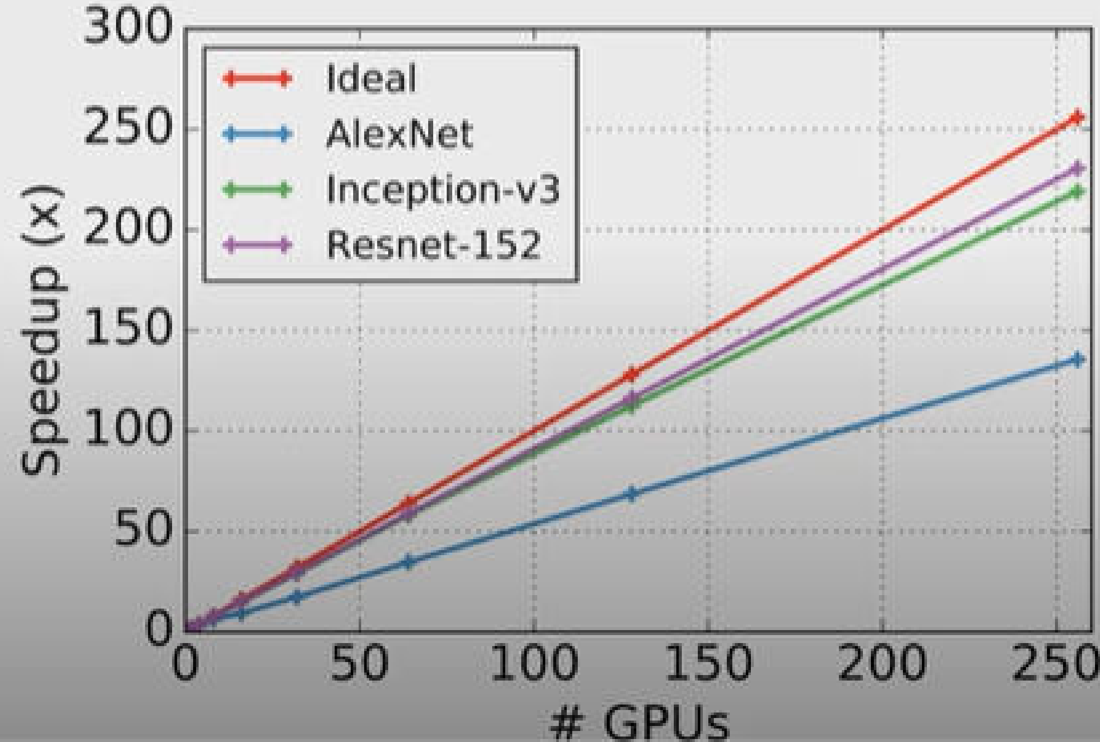
Demo - Matrix Factorization Using Your Own Code and the Amazon SageMaker Factorization Machine

DataSet

User Id	Movie Id	Rating	Timestamp
1	1	5	874965758
1	2	3	876893171
1	3	4	878542960
1	4	3	876893119
1	5	3	889751712

MXNet

- We are using MXNet and Gluon for coding.
- MXNet benchmark shows near linear performance across multiple machines.
- Gluon is a Pytorch-like imperative API for MXNet.



Amazon SageMaker

aws training and certification

- A fully managed service that enables data scientists and developers to quickly and easily build machine-learning based models into production smart applications.
- Amazon SageMaker includes several built-in state of the art algorithms that are fine-tuned to run on distributed environments. https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html
- Amongst built-in algorithms there is a Factorization Machine algorithm implemented using MXNet.
- All you need to do it to tune model's hyperparameters and passing your data to the model.

jupyter LinearMF Last Checkpoint: 4 hours ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted conda_mxnet_p36

Run Code

code in part inspired by: https://github.com/EthanRosenthal/torchmf

In [19]:
```
import os
import mxnet as mx
from mxnet import gluon, nd, ndarray

import pandas as pd
import numpy as np
```

In [20]:
```
data_path = 'ml-100k/'
num_emb = 64
opt = 'Adam'
lr = 0.02
mmntm = 0.
wd = 0.
batch_size = 50
ctx = mx.gpu(4)
```

In [3]:
```
def download_ml_data(prefix):
    if not os.path.exists("%s.zip" % prefix):
        print("Downloading MovieLens data: %s" % prefix)
        os.system("wget http://files.grouplens.org/datasets/movielens/%s.zip" % prefix)
        os.system("unzip %s.zip" % prefix)
```

In [4]:
```
download_ml_data('data')
```

```
Downloading MovieLens data: data
```

In [5]:
```
def max_id(fname):
    mu = 0
    mi = 0
    with open(fname) as f:
        for line in f:
            tks = line.strip().split('\t')
            if len(tks) != 4:
                continue
            mu = max(mu, int(tks[0]))
            mi = max(mi, int(tks[1]))
    return mu + 1, mi + 1
max_users, max_items = max_id(data_path + 'u.data')
```

In [6]:
```
train_df = pd.read_csv(data_path+'u1.base', header=None, sep='\t')
test_df = pd.read_csv(data_path+'u1.test', header=None, sep='\t')

train_data = nd.array(train_df[[0,1]].values, dtype=np.float32)
```

In [19]:
```
import os
import mxnet as mx
from mxnet import gluon, nd, ndarray

import pandas as pd
import numpy as np
```

In [20]:
```
data_path = 'ml-100k/'
num_emb = 64
opt = 'Adam'
lr = 0.02
mmntm = 0.
wd = 0.
batch_size = 50
ctx = mx.gpu(4)
```

In [3]:
```
def download_ml_data(prefix):
    if not os.path.exists("%s.zip" % prefix):
        print("Downloading MovieLens data: %s" % prefix)
        os.system("wget http://files.grouplens.org/datasets/movielens/%s.zip" % prefix)
        os.system("unzip %s.zip" % prefix)
```

In [4]:
```
download_ml_data('data')
```

```
Downloading MovieLens data: data
```

In [5]:
```
def max_id(fname):
    mu = 0
    mi = 0
    with open(fname) as f:
        for line in f:
            tks = line.strip().split('\t')
            if len(tks) != 4:
                continue
            mu = max(mu, int(tks[0]))
            mi = max(mi, int(tks[1]))
    return mu + 1, mi + 1
max_users, max_items = max_id(data_path + 'u.data')
```

```
        return mu + 1, mi + 1
max_users, max_items = max_id(data_path + 'u.data')
```

In [6]:
```
train_df = pd.read_csv(data_path+'u1.base', header=None, sep='\t')
test_df = pd.read_csv(data_path+'u1.test', header=None, sep='\t')

train_data = nd.array(train_df[[0,1]].values, dtype=np.float32)
train_label = nd.array(train_df[2].values, dtype=np.float32)

test_data = nd.array(test_df[[0,1]].values, dtype=np.float32)
test_label = nd.array(test_df[2].values, dtype=np.float32)
```

In [7]:
```
class SparseMatrixDataset(gluon.data.Dataset):
    def __init__(self, data, label):
        assert data.shape[0] == len(label)
        self.data = data
        self.label = label
        if isinstance(label, ndarray.NDArray) and len(label.shape) == 1:
            self._label = label.asnumpy()
        else:
            self._label = label

    def __getitem__(self, idx):
        return self.data[idx, 0], self.data[idx, 1], self.label[idx]

    def __len__(self):
        return self.data.shape[0]
```

In [8]:
```
class MFBlock(gluon.Block):
    def __init__(self, max_users, max_items, num_emb, dropout_p=0.5):
        super(MFBlock, self).__init__()

        self.max_users = max_users
        self.max_items = max_items
        self.dropout_p = dropout_p
        self.num_emb = num_emb

        with self.name_scope():
            self.user_embeddings = gluon.nn.Embedding(max_users, num_emb)
            self.item_embeddings = gluon.nn.Embedding(max_items, num_emb)
```

```
    return mu + 1, mi + 1
max_users, max_items = max_id(data_path + 'u.data')
```

In [6]:
```
train_df = pd.read_csv(data_path+'u1.base', header=None, sep='\t')
test_df = pd.read_csv(data_path+'u1.test', header=None, sep='\t')

train_data = nd.array(train_df[[0,1]].values, dtype=np.float32)
train_label = nd.array(train_df[2].values, dtype=np.float32)

test_data = nd.array(test_df[[0,1]].values, dtype=np.float32)
test_label = nd.array(test_df[2].values, dtype=np.float32)
```

In [7]:
```
class SparseMatrixDataset(gluon.data.Dataset):
    def __init__(self, data, label):
        assert data.shape[0] == len(label)
        self.data = data
        self.label = label
        if isinstance(label, ndarray.NDArray) and len(label.shape) == 1:
            self._label = label.asnumpy()
        else:
            self._label = label

    def __getitem__(self, idx):
        return self.data[idx, 0], self.data[idx, 1], self.label[idx]

    def __len__(self):
        return self.data.shape[0]
```

In [8]:
```
class MFBlock(gluon.Block):
    def __init__(self, max_users, max_items, num_emb, dropout_p=0.5):
        super(MFBlock, self).__init__()

        self.max_users = max_users
        self.max_items = max_items
        self.dropout_p = dropout_p
        self.num_emb = num_emb

        with self.name_scope():
            self.user_embeddings = gluon.nn.Embedding(max_users, num_emb)
            self.item_embeddings = gluon.nn.Embedding(max_items, num_emb)
```

```python
    def __init__(self, data, label):
        assert data.shape[0] == len(label)
        self.data = data
        self.label = label
        if isinstance(label, ndarray.NDArray) and len(label.shape) == 1:
            self._label = label.asnumpy()
        else:
            self._label = label

    def __getitem__(self, idx):
        return self.data[idx, 0], self.data[idx, 1], self.label[idx]

    def __len__(self):
        return self.data.shape[0]
```

In [8]:
```python
class MFBlock(gluon.Block):
    def __init__(self, max_users, max_items, num_emb, dropout_p=0.5):
        super(MFBlock, self).__init__()

        self.max_users = max_users
        self.max_items = max_items
        self.dropout_p = dropout_p
        self.num_emb = num_emb

        with self.name_scope():
            self.user_embeddings = gluon.nn.Embedding(max_users, num_emb)
            self.item_embeddings = gluon.nn.Embedding(max_items, num_emb)
            self.dropout = gluon.nn.Dropout(dropout_p)

    def forward(self, users, items):
        a = self.user_embeddings(users)
        b = self.item_embeddings(items)
        predictions = self.dropout(a) * self.dropout(b)
        predictions = nd.sum(predictions, axis=1)
        return predictions
```

In [9]:
```python
net = MFBlock(max_users=max_users, max_items=max_items, num_emb=num_emb, dropout_p=0.)
net.collect_params()
```

In [11]:
```
net.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx, force_reinit=True)
```

In [12]:
```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'learning_rate': lr, 'wd': wd, 'momentum': 0.9})
```

In [13]:
```
train_data_iter = gluon.data.DataLoader(SparseMatrixDataset(train_data, train_label),
                                        shuffle=True, batch_size=batch_size)
test_data_iter = gluon.data.DataLoader(SparseMatrixDataset(test_data, test_label),
                                        shuffle=True, batch_size=batch_size)
```

In [14]:
```
def eval_net(data, net):
    acc = mx.metric.RMSE()
    for i, (user, item, label) in enumerate(data):
        user = user.as_in_context(ctx).reshape((batch_size,))
        item = item.as_in_context(ctx).reshape((batch_size,))
        label = label.as_in_context(ctx).reshape((batch_size,))
        predictions = net(user, item)
        loss = loss_function(predictions, label)
        acc.update(preds=predictions, labels=label)
    return acc.get()[1]
```

In [15]:
```
eval_net(test_data_iter, net)
```

Out[15]: 3.5358702216744424

In [16]:
```
epochs = 10
#smoothing_constant = 10

def train(data_iter, net):
    a = []
    b = []
    c = []
    d = []
    for e in range(epochs):
        print("epoch: {}".format(e))
        for i, (user, item, label) in enumerate(train_data_iter):
            user = user.as_in_context(ctx).reshape((batch_size,))
```

```python
        loss = loss_function(predictions, label)
        acc.update(preds=predictions, labels=label)
    return acc.get()[1]
```

In [15]:

```python
eval_net(test_data_iter, net)
```

Out[15]: 3.5358702216744424

In [16]:

```python
epochs = 10
#smoothing_constant = 10

def train(data_iter, net):
    a = []
    b = []
    c = []
    d = []
    for e in range(epochs):
        print("epoch: {}".format(e))
        for i, (user, item, label) in enumerate(train_data_iter):
            user = user.as_in_context(ctx).reshape((batch_size,))
            item = item.as_in_context(ctx).reshape((batch_size,))
            label = label.as_in_context(ctx).reshape((batch_size,))
            with mx.autograd.record():
                output = net(user, item)
                loss = loss_function(output, label)
                loss.backward()
            net.collect_params().values()
            trainer.step(batch_size)
        a = eval_net(test_data_iter, net)
        b = eval_net(train_data_iter, net)
        print("EPOCH {}: RMSE ON TRAINING and TEST: {}. {}".format(e,a,b))

    return a, b
```

In [17]:

```python
(a,b) = train(train_data_iter, net)
```

```
epoch: 0
EPOCH 0: RMSE ON TRAINING and TEST: 3.5338459888896489. 3.523215442742407
epoch: 1
EPOCH 1: RMSE ON TRAINING and TEST: 3.398607304608822. 3.328483776437491
```

```
            loss.backward()
            net.collect_params().values()
            trainer.step(batch_size)
        a = eval_net(test_data_iter, net)
        b = eval_net(train_data_iter, net)
        print("EPOCH {}: RMSE ON TRAINING and TEST: {}. {}".format(e,a,b))

    return a, b
```

In [17]:
```
(a,b) = train(train_data_iter, net)
```

```
epoch: 0
EPOCH 0: RMSE ON TRAINING and TEST: 3.533845988896489. 3.523215442742407
epoch: 1
EPOCH 1: RMSE ON TRAINING and TEST: 3.398607304608822. 3.328483776437491
epoch: 2
EPOCH 2: RMSE ON TRAINING and TEST: 2.032065240380168. 1.797713072796911
epoch: 3
EPOCH 3: RMSE ON TRAINING and TEST: 1.3171076374143362. 1.1745245898365975
epoch: 4
EPOCH 4: RMSE ON TRAINING and TEST: 1.0678859624534844. 0.9613828420139849
epoch: 5
EPOCH 5: RMSE ON TRAINING and TEST: 0.9528403462797403. 0.866311743491143
epoch: 6
EPOCH 6: RMSE ON TRAINING and TEST: 0.8905037337005138. 0.8156895110182464
epoch: 7
EPOCH 7: RMSE ON TRAINING and TEST: 0.8528216697186232. 0.785909748146683
epoch: 8
EPOCH 8: RMSE ON TRAINING and TEST: 0.8289129304856062. 0.7666749547488988
epoch: 9
EPOCH 9: RMSE ON TRAINING and TEST: 0.8108262846082449. 0.7518034620203078
```

In [18]:
```
(a,b)
```

Out[18]:
```
(0.8108262846082449, 0.7518034620203078)
```

In [ ]:

39:22 / 54:31

In [7]:
```python
class MFBlock(gluon.Block):
    def __init__(self, max_users, max_items, num_emb, dropout_p=0.5):
        super(MFBlock, self).__init__()

        self.max_users = max_users
        self.max_items = max_items
        self.dropout_p = dropout_p
        self.num_emb = num_emb

        with self.name_scope():
            self.user_embeddings = gluon.nn.Embedding(max_users, num_emb)
            self.item_embeddings = gluon.nn.Embedding(max_items, num_emb)
            self.dropout = gluon.nn.Dropout(dropout_p)
            self.dense = gluon.nn.Dense(num_emb, activation='relu')

    def forward(self, users, items):
        a = self.user_embeddings(users)
        a = self.dense(a)

        b = self.item_embeddings(items)
        b = self.dense(b)

        predictions = self.dropout(a) * self.dropout(b)
        predictions = nd.sum(predictions, axis=1)
        return predictions
```

In [8]:
```python
net = MFBlock(max_users=max_users, max_items=max_items, num_emb=num_emb, dropout_p=0.)
net.collect_params()
```

Out[8]:
```
mfblock0_ (
  Parameter mfblock0_embedding0_weight (shape=(944, 64), dtype=<class 'numpy.float32'>)
  Parameter mfblock0_embedding1_weight (shape=(1683, 64), dtype=<class 'numpy.float32'>)
  Parameter mfblock0_dense0_weight (shape=(64, 0), dtype=<class 'numpy.float32'>)
```

```
                loss = loss_function(output, label)
                loss.backward()
            net.collect_params().values()
            trainer.step(batch_size)
        print("EPOCH {}: RMSE ON TRAINING and TEST: {}. {}".format(e,
                                                                    eval_net(train_data_iter, net),
                                                                    eval_net(test_data_iter, net)))
    return "end of training"
```

In [16]:

```
train(train_data_iter, net)
```

```
epoch: 0
EPOCH 0: RMSE ON TRAINING and TEST: 0.7461072485804557. 0.7763755543172359
epoch: 1
EPOCH 1: RMSE ON TRAINING and TEST: 0.7369058181449771. 0.7680148655653
epoch: 2
EPOCH 2: RMSE ON TRAINING and TEST: 0.7472432709142566. 0.7772404563993216
epoch: 3
EPOCH 3: RMSE ON TRAINING and TEST: 0.7370284162349999. 0.7691778198421001
epoch: 4
EPOCH 4: RMSE ON TRAINING and TEST: 0.7406699358060956. 0.7754190125107765
epoch: 5
EPOCH 5: RMSE ON TRAINING and TEST: 0.7273183228254319. 0.7669575016319752
epoch: 6
EPOCH 6: RMSE ON TRAINING and TEST: 0.7240261309757828. 0.7765023551046848
epoch: 7
EPOCH 7: RMSE ON TRAINING and TEST: 0.693246350326389. 0.7645233050823211
epoch: 8
EPOCH 8: RMSE ON TRAINING and TEST: 0.6648808738991618. 0.7582760076761246
epoch: 9
EPOCH 9: RMSE ON TRAINING and TEST: 0.6372081472031772. 0.7497557436436415
```

Out[16]: 'end of training'

In [17]:

```
net1 = gluon.nn.Sequential()
with net1.name_scope():
    net1.add(gluon.nn.Embedding(max_users, num_emb))
    net1.add(gluon.nn.Dense(64))
```

```python
from scipy.sparse import lil_matrix

BUCKET = 'cyrusmv-sagemaker-demos'
s3 = boto3.client('s3')


def download_file(s3_source, dest):
    if not os.path.exists(dest):
        os.makedirs(dest)

    url = urlparse(s3_source)
    bucket, key = url.netloc, url.path.lstrip('/')
    file_name = key.split('/')[-1]
    with open('%s/%s' % (dest,file_name), 'wb') as data:
      s3.download_fileobj(bucket, key, data)


def loadDataset(filename, lines, columns):
    # Features are one-hot encoded in a sparse matrix
    X = lil_matrix((lines, columns)).astype('float32')
    # Labels are stored in a vector
    Y = []
    line=0
    with open(filename,'r') as f:
        samples=csv.reader(f,delimiter='\t')
        for userId,movieId,rating,timestamp in samples:
            X[line,int(userId)-1] = 1
            X[line,int(nbUsers)+int(movieId)-1] = 1
            Y.append(int(rating))
            line=line+1

    Y=np.array(Y).astype('float32')
    return X,Y

nbUsers=943
nbMovies=1682
```

```
<_io.BytesIO object at 0x7f9ac0385f50>
Wrote dataset: cyrusmv-sagemaker-demos/exercise4/fm-movielens100k/train/train.protobuf
<_io.BytesIO object at 0x7f9ac0343c50>
Wrote dataset: cyrusmv-sagemaker-demos/exercise4/fm-movielens100k/test/test.protobuf
Output: s3://cyrusmv-sagemaker-demos/exercise4/fm-movielens100k/output
```

In [3]:

```
import sagemaker
from sagemaker import get_execution_role

train_data = 's3://%s/exercise4/fm-movielens100k/train/train.protobuf' % BUCKET
test_data = 's3://%s/exercise4/fm-movielens100k/test/test.protobuf' % BUCKET

containers = {'us-west-2': '174872318107.dkr.ecr.us-west-2.amazonaws.com/factorization-machines:latest',
              'us-east-1': '382416733822.dkr.ecr.us-east-1.amazonaws.com/factorization-machines:latest',
              'us-east-2': '404615174143.dkr.ecr.us-east-2.amazonaws.com/factorization-machines:latest',
              'eu-west-1': '438346466558.dkr.ecr.eu-west-1.amazonaws.com/factorization-machines:latest'}

fm = sagemaker.estimator.Estimator(containers[boto3.Session().region_name],
                                   get_execution_role(),
                                   train_instance_count=1,
                                   train_instance_type='ml.c4.xlarge',
                                   output_path=output_prefix,
                                   sagemaker_session=sagemaker.Session())

fm.set_hyperparameters(feature_dim=nbFeatures,
                       predictor_type='regressor',
                       mini_batch_size=1000,
                       num_factors=64,
                       _speedometer_period=10,
                       epochs=50)

fm.fit({'train': train_data, 'test': test_data})
```


er", "Operation": "training", "Algorithm": "factorization-machines", "epoch": 33}, "StartTime": 1521675975.624592}

42:03 / 54:31

```
                    _speedometer_period=10,
                    epochs=50)

fm.fit({'train': train_data, 'test': test_data})
```

```
[03/21/2018 23:46:18 INFO 139926754449216] Epoch[1] Batch [70]#011Speed: 152590.06 samples/sec#011rmse=0.963353
```

#metrics {"Metrics": {"update.time": {"count": 1, "max": 500.90694427490234, "sum": 500.90694427490234, "min": 500.90694427490234}}, "EndTime": 1521675978.700887, "Dimensions": {"Host": "algo-1", "Operation": "training", "Algorithm": "factorization-machines"}, "StartTime": 1521675978.19969}

#metrics {"Metrics": {"Max Batches Seen Between Resets": {"count": 1, "max": 80, "sum": 80.0, "min": 80}, "Number of Batches Since Last Reset": {"count": 1, "max": 80, "sum": 80.0, "min": 80}, "Number of Records Since Last Reset": {"count": 1, "max": 80000, "sum": 80000.0, "min": 80000}, "Total Batches Seen": {"count": 1, "max": 3201, "sum": 3201.0, "min": 3201}, "Total Records Seen": {"count": 1, "max": 3201000, "sum": 3201000.0, "min": 3201000}, "Max Records Seen Between Resets": {"count": 1, "max": 80000, "sum": 80000.0, "min": 80000}, "Reset Count": {"count": 1, "max": 41, "sum": 41.0, "min": 41}}, "EndTime": 1521675978.701077, "Dimensions": {"Host": "algo-1", "Meta": "training_data_iter", "Operation": "training", "Algorithm": "factorization-machines", "epoch": 39}, "StartTime": 1521675978.701025}

```
[03/21/2018 23:46:18 INFO 139926754449216] Epoch[1] Batch [10]#011Speed: 144500.13 samples/sec#011rmse=0.999548
[03/21/2018 23:46:18 INFO 139926754449216] Epoch[1] Batch [20]#011Speed: 158260.69 samples/sec#011rmse=0.974219
[03/21/2018 23:46:18 INFO 139926754449216] Epoch[1] Batch [30]#011Speed: 146147.58 samples/sec#011rmse=0.982746
[03/21/2018 23:46:18 INFO 139926754449216] Epoch[1] Batch [40]#011Speed: 144542.45 samples/sec#011rmse=0.973204
[03/21/2018 23:46:19 INFO 139926754449216] Epoch[1] Batch [50]#011Speed: 150276.38 samples/sec#011rmse=0.971157
[03/21/2018 23:46:19 INFO 139926754449216] Epoch[1] Batch [60]#011Speed: 148270.97 samples/sec#011rmse=0.964169
```

In [4]:

```
fm_predictor = fm.deploy(instance_type='ml.c4.xlarge', initial_instance_count=1)
```

```
INFO:sagemaker:Creating model with name: factorization-machines-2018-03-21-23-47-57-115
INFO:sagemaker:Creating endpoint with name factorization-machines-2018-03-21-23-40-14-697
```

```
---------------------------------------------------------------------------------------!
```

In [5]:

```
import json
import numpy as np
from sagemaker.predictor import json_deserializer
```

43:33 / 54:31

```
                    _speedometer_period=10,
                    epochs=50)

fm.fit({'train': train_data, 'test': test_data})
```

```
[03/21/2018 23:46:23 INFO 139926754449216] Saved checkpoint to "/tmp/tmpTlZMG_/state-0001.params"
[03/21/2018 23:46:23 INFO 139926754449216] #test_score (algo-1) : rmse
[03/21/2018 23:46:23 INFO 139926754449216] #test_score (algo-1) : 1.00061402047
```

#metrics {"Metrics": {"Max Batches Seen Between Resets": {"count": 1, "max": 20, "sum": 20.0, "min": 20}, "Number of Batches Since Last Reset": {"count": 1, "max": 20, "sum": 20.0, "min": 20}, "Number of Records Since Last Reset": {"count": 1, "max": 20000, "sum": 20000.0, "min": 20000}, "Total Batches Seen": {"count": 1, "max": 20, "sum": 20.0, "min": 20}, "Total Records Seen": {"count": 1, "max": 20000, "sum": 20000.0, "min": 20000}, "Max Records Seen Between Resets": {"count": 1, "max": 20000, "sum": 20000.0, "min": 20000}, "Reset Count": {"count": 1, "max": 1, "sum": 1.0, "min": 1}}, "EndTime": 1521675983.980208, "Dimensions": {"Host": "algo-1", "Meta": "test_data_iter", "Operation": "training", "Algorithm": "factorization-machines"}, "StartTime": 1521675983.980173}

#metrics {"Metrics": {"totaltime": {"count": 1, "max": 26234.050989151, "sum": 26234.050989151, "min": 26234.050989151}, "setuptime": {"count": 1, "max": 40.194034576416016, "sum": 40.194034576416016, "min": 40.194034576416016}}, "EndTime": 1521675983.981656, "Dimensions": {"Host": "algo-1", "Operation": "training", "Algorithm": "factorization-machines"}, "StartTime": 1521675983.902766}

```
===== Job Complete =====
Billable seconds: 217
```

In [4]:
```
fm_predictor = fm.deploy(instance_type='ml.c4.xlarge', initial_instance_count=1)
```

```
INFO:sagemaker:Creating model with name: factorization-machines-2018-03-21-23-47-57-115
INFO:sagemaker:Creating endpoint with name factorization-machines-2018-03-21-23-40-14-697
```

```
---------------------------------------------------------------------------------!
```

In [5]:
```
import json
import numpy as np
from sagemaker.predictor import json_deserializer
```

In [5]:

```python
import json
import numpy as np
from sagemaker.predictor import json_deserializer

nbUsers=943
nbMovies=1682
nbFeatures=nbUsers+nbMovies

def fm_serializer(data):
    js = {'instances': []}
    for row in data:
        keys = np.argwhere(row == np.amax(row)).flatten().tolist()
        js['instances'].append({
            'data':{
                'features': {
                    'keys': keys,
                    'shape': [nbFeatures],
                    'values': [1]*len(keys)
                }
            }
            })
    #print js
    return json.dumps(js)

fm_predictor.content_type = 'application/json'
fm_predictor.serializer = fm_serializer
fm_predictor.deserializer = json_deserializer


result = fm_predictor.predict(X_test[1000:1010].toarray())
print(result)
print()
print (Y_test[1000:1010])
```

```
def fm_serializer(data):
    js = {'instances': []}
    for row in data:
        keys = np.argwhere(row == np.amax(row)).flatten().tolist()
        js['instances'].append({
            'data':{
                'features': {
                    'keys': keys,
                    'shape': [nbFeatures],
                    'values': [1]*len(keys)
                }
            }
            })
    #print js
    return json.dumps(js)

fm_predictor.content_type = 'application/json'
fm_predictor.serializer = fm_serializer
fm_predictor.deserializer = json_deserializer


result = fm_predictor.predict(X_test[1000:1010].toarray())
print(result)
print()
print (Y_test[1000:1010])
```

```
{u'predictions': [{u'score': 3.3320837020874023}, {u'score': 3.0627427101135254}, {u'score': 3.305492639541626}, {u's
core': 2.9380016326904297}, {u'score': 2.8458235263824463}, {u'score': 3.073624849319458}, {u'score': 3.0407218933105
47}, {u'score': 3.3230855464935303}, {u'score': 3.044969081878662}, {u'score': 3.535712480545044}]}
()
[2. 1. 3. 3. 3. 1. 3. 3. 1. 4.]
```

45:07 / 54:31

Factorization Machines Hyperparameters

Parameter Name	Description
`feature_dim`	Dimension of the input feature space. This could be very high with sparse input. Required. Valid values: Positive integer. Suggested value range: [10000,10000000] Default value: -
`num_factors`	Dimensionality of factorization. Required. Valid values: Positive integer. Suggested value range: [2,1000] Default value: -
`predictor_type`	Type of predictor. Required. Valid values: String: `binary_classifier` or regressor Default value: -
`mini_batch_size`	Size of mini-batch used for training. Valid values: positive integer Default value: 1000
`epochs`	Number of training epochs to run. Valid values: positive integer Default value: 1
`clip_gradient`	Optimizer parameter. Clip the gradient by projecting onto the box [`-clip_gradient`, `+clip_gradient`]. Valid values: float Default value: -
`eps`	Optimizer parameter. Small value to avoid division by 0. Valid values: float Default value:

`predictor_type`	Type of predictor. Required. Valid values: String: `binary_classifier` or regressor Default value: -
`mini_batch_size`	Size of mini-batch used for training. Valid values: positive integer Default value: 1000
`epochs`	Number of training epochs to run. Valid values: positive integer Default value: 1
`clip_gradient`	Optimizer parameter. Clip the gradient by projecting onto the box [`-clip_gradient`, `+clip_gradient`]. Valid values: float Default value: -
`eps`	Optimizer parameter. Small value to avoid division by 0. Valid values: float Default value: -
`rescale_grad`	Optimizer parameter. If set, multiplies the gradient with `rescale_grad` before updating. Often choose to be 1.0/`batch_size`. Valid values: float Default value: -
`bias_lr`	Learning rate for the bias term. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.1
`linear_lr`	Learning rate for linear terms. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.001

	Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.1
`linear_lr`	Learning rate for linear terms. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.001
`factors_lr`	Learning rate for factorization terms. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.0001
`bias_wd`	Weight decay for the bias term. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.01
`linear_wd`	Weight decay for linear terms. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.001
`factors_wd`	Weight decay for factorization terms. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.00001
`bias_init_method`	Initialization method for the bias term. • *normal*: Initializes weights with random values sampled from a normal distribution with a mean of zero and standard deviation specified by `bias_init_sigma`. • *uniform*: Initializes weights with random values uniformly sampled from a range specified by [-`bias_init_scale`, +`bias_init_scale`]. • *constant*: Initializes the weights to a scalar value specified by `bias_init_value`. Valid values: *uniform*, *normal*, or *constant* Default value: *normal*
`bias_init_scale`	Range for initialization of the bias term. Takes effect if `bias_init_method` is set to *uniform*.

	Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.01
`linear_wd`	Weight decay for linear terms. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.001
`factors_wd`	Weight decay for factorization terms. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.00001
`bias_init_method`	Initialization method for the bias term. • *normal*: Initializes weights with random values sampled from a normal distribution with a mean of zero and standard deviation specified by `bias_init_sigma`. • *uniform*: Initializes weights with random values uniformly sampled from a range specified by [`-bias_init_scale`, `+bias_init_scale`]. • *constant*: Initializes the weights to a scalar value specified by `bias_init_value`. Valid values: *uniform*, *normal*, or *constant* Default value: *normal*
`bias_init_scale`	Range for initialization of the bias term. Takes effect if `bias_init_method` is set to *uniform*. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: -
`bias_init_sigma`	Standard deviation for initialization of the bias term. Takes effect if `bias_init_method` is set to *normal*. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.01
`bias_init_value`	Initial value of the bias term. Takes effect if `bias_init_method` is set to *constant*. Valid values: Float. Suggested value range: [1e-8, 512]/ Default value: -
`linear_init_method`	Initialization method for linear terms.

Amazon SageMaker
Developer Guide
Documentation - This Guide
Search
What Is Amazon SageMaker?
How It Works
Getting Started
Using Built-in Algorithms
Common Information
Linear Learner
Factorization Machines
How It Works
Hyperparameters
Inference Formats
XGBoost Algorithm
Image Classification Algorithm
Sequence to Sequence (seq2seq)
K-Means Algorithm
Principal Component Analysis (PCA)
Latent Dirichlet Allocation (LDA)
Neural Topic Model (NTM)
DeepAR Forecasting
BlazingText
Random Cut Forest
Using Your Own Algorithms

`bias_init_method`	Initialization method for the bias term. • *normal*: Initializes weights with random values sampled from a normal distribution with a mean of zero and standard deviation specified by `bias_init_sigma`. • *uniform*: Initializes weights with random values uniformly sampled from a range specified by [`-bias_init_scale`, `+bias_init_scale`]. • *constant*: Initializes the weights to a scalar value specified by `bias_init_value`. Valid values: *uniform*, *normal*, or *constant* Default value: *normal*
`bias_init_scale`	Range for initialization of the bias term. Takes effect if `bias_init_method` is set to *uniform*. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: -
`bias_init_sigma`	Standard deviation for initialization of the bias term. Takes effect if `bias_init_method` is set to *normal*. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.01
`bias_init_value`	Initial value of the bias term. Takes effect if `bias_init_method` is set to *constant*. Valid values: Float. Suggested value range: [1e-8, 512]. Default value: -
`linear_init_method`	Initialization method for linear terms. • *normal* Initializes weights with random values sampled from a normal distribution with a mean of zero and standard deviation specified by `linear_init_sigma`. • *uniform* Initializes weights with random values uniformly sampled from a range specified by [`-linear_init_scale`, `+linear_init_scale`]. • *constant* Initializes the weights to a scalar value specified by `linear_init_value`. Valid values: *uniform*, *normal*, or *constant*. Default value: *normal*
`linear_init_scale`	Range for initialization of linear terms. Takes effect if `linear_init_method` is set to *uniform*. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: -

	• *constant*: Initializes the weights to a scalar value specified by `bias_init_value`. Valid values: *uniform*, *normal*, or *constant* Default value: *normal*
`bias_init_scale`	Range for initialization of the bias term. Takes effect if `bias_init_method` is set to *uniform*. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: -
`bias_init_sigma`	Standard deviation for initialization of the bias term. Takes effect if `bias_init_method` is set to *normal*. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.01
`bias_init_value`	Initial value of the bias term. Takes effect if `bias_init_method` is set to *constant*. Valid values: Float. Suggested value range: [1e-8, 512]/ Default value: -
`linear_init_method`	Initialization method for linear terms. • *normal* Initializes weights with random values sampled from a normal distribution with a mean of zero and standard deviation specified by `linear_init_sigma`. • *uniform* Initializes weights with random values uniformly sampled from a range specified by [-`linear_init_scale`, +`linear_init_scale`]. • *constant* Initializes the weights to a scalar value specified by `linear_init_value`. Valid values: *uniform*, *normal*, or *constant*. Default value: *normal*
`linear_init_scale`	Range for initialization of linear terms. Takes effect if `linear_init_method` is set to *uniform*. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: -
`linear_init_sigma`	Standard deviation for initialization of linear terms. Takes effect if `linear_init_method` is set to *normal*. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.01

	• *constant*: Initializes the weights to a scalar value specified by `bias_init_value`. Valid values: *uniform*, *normal*, or *constant* Default value: *normal*
`bias_init_scale`	Range for initialization of the bias term. Takes effect if `bias_init_method` is set to *uniform*. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: -
`bias_init_sigma`	Standard deviation for initialization of the bias term. Takes effect if `bias_init_method` is set to *normal*. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.01
`bias_init_value`	Initial value of the bias term. Takes effect if `bias_init_method` is set to *constant*. Valid values: Float. Suggested value range: [1e-8, 512]. Default value: -
`linear_init_method`	Initialization method for linear terms. • *normal* Initializes weights with random values sampled from a normal distribution with a mean of zero and standard deviation specified by `linear_init_sigma`. • *uniform* Initializes weights with random values uniformly sampled from a range specified by [-`linear_init_scale`, +`linear_init_scale`]. • *constant* Initializes the weights to a scalar value specified by `linear_init_value`. Valid values: *uniform*, *normal*, or *constant*. Default value: *normal*
`linear_init_scale`	Range for initialization of linear terms. Takes effect if `linear_init_method` is set to *uniform*. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: -
`linear_init_sigma`	Standard deviation for initialization of linear terms. Takes effect if `linear_init_method` is set to *normal*. Valid values: Non-negative float. Suggested value range: [1e-8, 512]. Default value: 0.01

Development and Deployment

aws training and certification

- Loss function is one of the most important areas to pay attention to.
- Multi-label cross-entropy loss has worked well in the past.
- This is relatively simple to apply to a wide variety of model types. Ranking loss often works better, but is more complex to apply correctly.
- Scalability is always a big challenge. Offline batch computation and saving the results can help.

Logging and Measurement

- Deploying a recommender system requires some care since a model only succeeds if good behavioral data can be logged.
- Moreover, without good logging it is impossible to assess the quality of the deployment. Tools such as Amazon Kinesis are ideally suited for this purpose.
- Display bias is very strong – this means that customers are more likely to click on a mediocre recommendation that they see than an excellent recommendation they don't see.