Algorithm/DeepFM/main.py

if \_\_name\_\_ == '\_\_main\_\_':

import Recommender\_System.utility.gpu\_memory\_growth

from Recommender\_System.data import data\_loader, data\_process

from Recommender\_System.algorithm.DeepFM.model import DeepFM\_model

from Recommender\_System.algorithm.train import train

n\_user, n\_item, train\_data, test\_data, topk\_data = data\_process.pack(data\_loader.ml100k)

model = DeepFM\_model(n\_user, n\_item, dim=8, layers=[16, 16, 16], l2=1e-5)

train(model, train\_data, test\_data, topk\_data, epochs=10)

algorithm/DeepFM/model.py

import tensorflow as tf

from Recommender\_System.utility.decorator import logger

@logger('初始化DeepFM模型：', ('n\_user', 'n\_item', 'dim', 'layers', 'l2'))

def DeepFM\_model(n\_user: int, n\_item: int, dim=8, layers=[16, 16, 16], l2=1e-6) -> tf.keras.Model:

l2 = tf.keras.regularizers.l2(l2)

user\_id = tf.keras.Input(shape=(), name='user\_id', dtype=tf.int32)

user\_embedding = tf.keras.layers.Embedding(n\_user, dim, embeddings\_regularizer=l2)(user\_id)

item\_id = tf.keras.Input(shape=(), name='item\_id', dtype=tf.int32)

item\_embedding = tf.keras.layers.Embedding(n\_item, dim, embeddings\_regularizer=l2)(item\_id)

user\_bias = tf.keras.layers.Embedding(n\_user, 1, embeddings\_initializer='zeros')(user\_id)

item\_bias = tf.keras.layers.Embedding(n\_item, 1, embeddings\_initializer='zeros')(item\_id)

fm = tf.reduce\_sum(user\_embedding \* item\_embedding, axis=1, keepdims=True) + user\_bias + item\_bias

deep = tf.concat([user\_embedding, item\_embedding], axis=1)

for layer in layers:

deep = tf.keras.layers.Dense(layer, activation='relu', kernel\_regularizer=l2)(deep)

deep = tf.keras.layers.Dense(1, kernel\_regularizer=l2)(deep)

out = tf.keras.activations.sigmoid(fm + deep)

return tf.keras.Model(inputs=[user\_id, item\_id], outputs=out)

if \_\_name\_\_ == '\_\_main\_\_':

tf.keras.utils.plot\_model(DeepFM\_model(1, 1), 'graph.png', show\_shapes=True)

algorithm/FM/main.py

if \_\_name\_\_ == '\_\_main\_\_':

import Recommender\_System.utility.gpu\_memory\_growth

from Recommender\_System.data import data\_loader, data\_process

from Recommender\_System.algorithm.FM.model import FM\_model

from Recommender\_System.algorithm.train import train

n\_user, n\_item, train\_data, test\_data, topk\_data = data\_process.pack(data\_loader.ml100k)

model = FM\_model(n\_user, n\_item, dim=16, l2=1e-6)

train(model, train\_data, test\_data, topk\_data, epochs=10, batch=512)

algorithm/FM/model.py

import tensorflow as tf

from Recommender\_System.utility.decorator import logger

@logger('初始化FM模型：', ('n\_user', 'n\_item', 'dim', 'l2'))

def FM\_model(n\_user: int, n\_item: int, dim=8, l2=1e-6) -> tf.keras.Model:

l2 = tf.keras.regularizers.l2(l2)

user\_id = tf.keras.Input(shape=(), name='user\_id', dtype=tf.int32)

user\_embedding = tf.keras.layers.Embedding(n\_user, dim, embeddings\_regularizer=l2)(user\_id)

user\_bias = tf.keras.layers.Embedding(n\_user, 1, embeddings\_initializer='zeros')(user\_id)

item\_id = tf.keras.Input(shape=(), name='item\_id', dtype=tf.int32)

item\_embedding = tf.keras.layers.Embedding(n\_item, dim, embeddings\_regularizer=l2)(item\_id)

item\_bias = tf.keras.layers.Embedding(n\_item, 1, embeddings\_initializer='zeros')(item\_id)

x = tf.reduce\_sum(user\_embedding \* item\_embedding, axis=1, keepdims=True) + user\_bias + item\_bias

out = tf.keras.activations.sigmoid(x)

return tf.keras.Model(inputs=[user\_id, item\_id], outputs=out)

if \_\_name\_\_ == '\_\_main\_\_':

tf.keras.utils.plot\_model(FM\_model(1, 1), 'graph.png', show\_shapes=True)

algorithm/KGCN/layer.py

from abc import abstractmethod

import tensorflow as tf

class Aggregator(tf.keras.layers.Layer):

def \_\_init\_\_(self, activation='relu', kernel\_regularizer=None, \*\*kwargs):

super(Aggregator, self).\_\_init\_\_(\*\*kwargs)

self.activation = tf.keras.activations.get(activation)

self.kernel\_regularizer = tf.keras.regularizers.get(kernel\_regularizer)

def call(self, inputs, \*\*kwargs):

self\_vectors, neighbor\_vectors, neighbor\_relations, user\_embeddings = inputs

\_, neighbor\_iter, dim = self\_vectors.shape

neighbor\_size = kwargs['neighbor\_size']

neighbor\_vectors = tf.reshape(neighbor\_vectors, shape=(-1, neighbor\_iter, neighbor\_size, dim))

neighbor\_relations = tf.reshape(neighbor\_relations, shape=(-1, neighbor\_iter, neighbor\_size, dim))

outputs = self.\_call(self\_vectors, neighbor\_vectors, neighbor\_relations, user\_embeddings, \*\*kwargs)

if self.activation is not None:

outputs = self.activation(outputs)

return outputs

@abstractmethod

def \_call(self, self\_vectors, neighbor\_vectors, neighbor\_relations, user\_embeddings, \*\*kwargs):

# self\_vectors: [batch, neighbor\_iter, dim]

# neighbor\_vectors: [batch, neighbor\_iter, neighbor\_size, dim]

# neighbor\_relations: [batch, neighbor\_iter, neighbor\_size, dim]

# user\_embeddings: [batch, dim]

pass

def \_mix\_neighbor\_vectors(self, neighbor\_vectors, neighbor\_relations, user\_embeddings):

dim = user\_embeddings.shape[-1]

avg = False

if not avg:

user\_embeddings = tf.reshape(user\_embeddings, shape=(-1, 1, 1, dim)) # [batch, 1, 1, dim]

user\_relation\_scores = tf.reduce\_mean(user\_embeddings \* neighbor\_relations, axis=-1) # [batch, neighbor\_iter, neighbor\_size]

user\_relation\_scores\_normalized = tf.nn.softmax(user\_relation\_scores, axis=-1) # [batch, neighbor\_iter, neighbor\_size]

user\_relation\_scores\_normalized = tf.expand\_dims(user\_relation\_scores\_normalized, axis=-1) # [batch, neighbor\_iter, neighbor\_size, 1]

neighbors\_aggregated = tf.reduce\_mean(user\_relation\_scores\_normalized \* neighbor\_vectors, axis=2) # [batch, neighbor\_iter, dim]

else:

neighbors\_aggregated = tf.reduce\_mean(neighbor\_vectors, axis=2) # [batch, neighbor\_iter, dim]

return neighbors\_aggregated

class SumAggregator(Aggregator):

def build(self, input\_shape):

dim = input\_shape[-1][-1]

self.kernel = self.add\_weight('kernel', shape=(dim, dim), initializer='glorot\_uniform', regularizer=self.kernel\_regularizer)

self.bias = self.add\_weight('bias', shape=(dim,), initializer='zeros')

def \_call(self, self\_vectors, neighbor\_vectors, neighbor\_relations, user\_embeddings, \*\*kwargs):

\_, neighbor\_iter, dim = self\_vectors.shape

neighbors\_agg = self.\_mix\_neighbor\_vectors(neighbor\_vectors, neighbor\_relations, user\_embeddings) # [batch, neighbor\_iter, dim]

output = tf.reshape(self\_vectors + neighbors\_agg, shape=(-1, dim)) # [batch \* neighbor\_iter, dim]

#if kwargs['training']:

# output = tf.nn.dropout(output, rate=0.2)

output = tf.nn.bias\_add(tf.matmul(output, self.kernel), self.bias) # [batch \* neighbor\_iter, dim]

return tf.reshape(output, shape=(-1, neighbor\_iter, dim)) # [batch, neighbor\_iter, dim]

class ConcatAggregator(Aggregator):

def build(self, input\_shape):

dim = input\_shape[-1][-1]

self.kernel = self.add\_weight('kernel', shape=(dim \* 2, dim), initializer='glorot\_uniform', regularizer=self.kernel\_regularizer)

self.bias = self.add\_weight('bias', shape=(dim,), initializer='zeros')

def \_call(self, self\_vectors, neighbor\_vectors, neighbor\_relations, user\_embeddings, \*\*kwargs):

\_, neighbor\_iter, dim = self\_vectors.shape

neighbors\_agg = self.\_mix\_neighbor\_vectors(neighbor\_vectors, neighbor\_relations, user\_embeddings) # [batch, neighbor\_iter, dim]

output = tf.concat([self\_vectors, neighbors\_agg], axis=2) # [batch, neighbor\_iter, dim \* 2]

output = tf.reshape(output, shape=(-1, dim \* 2)) # [batch \* neighbor\_iter, dim \* 2]

#if kwargs['training']:

# output = tf.nn.dropout(output, rate=0.2)

output = tf.nn.bias\_add(tf.matmul(output, self.kernel), self.bias) # [batch \* neighbor\_iter, dim]

return tf.reshape(output, shape=(-1, neighbor\_iter, dim)) # [batch, neighbor\_iter, dim]

class NeighborAggregator(Aggregator):

def build(self, input\_shape):

dim = input\_shape[-1][-1]

self.kernel = self.add\_weight('kernel', shape=(dim, dim), initializer='glorot\_uniform', regularizer=self.kernel\_regularizer)

self.bias = self.add\_weight('bias', shape=(dim,), initializer='zeros')

def \_call(self, self\_vectors, neighbor\_vectors, neighbor\_relations, user\_embeddings, \*\*kwargs):

\_, neighbor\_iter, dim = self\_vectors.shape

neighbors\_agg = self.\_mix\_neighbor\_vectors(neighbor\_vectors, neighbor\_relations, user\_embeddings) # [batch, neighbor\_iter, dim]

output = tf.reshape(neighbors\_agg, shape=(-1, dim)) # [batch \* neighbor\_iter, dim]

#if kwargs['training']:

# output = tf.nn.dropout(output, rate=0.2)

output = tf.nn.bias\_add(tf.matmul(output, self.kernel), self.bias) # [batch \* neighbor\_iter, dim]

return tf.reshape(output, shape=(-1, neighbor\_iter, dim)) # [batch, neighbor\_iter, dim]

algorithm/KGCN/main.py

if \_\_name\_\_ == '\_\_main\_\_':

import Recommender\_System.utility.gpu\_memory\_growth

from Recommender\_System.algorithm.KGCN.tool import construct\_undirected\_kg, get\_adj\_list

from Recommender\_System.algorithm.KGCN.model import KGCN\_model

from Recommender\_System.algorithm.KGCN.train import train

from Recommender\_System.data import kg\_loader, data\_process

import tensorflow as tf

n\_user, n\_item, n\_entity, n\_relation, train\_data, test\_data, kg, topk\_data = data\_process.pack\_kg(kg\_loader.ml1m\_kg1m, negative\_sample\_threshold=4)

neighbor\_size = 16

adj\_entity, adj\_relation = get\_adj\_list(construct\_undirected\_kg(kg), n\_entity, neighbor\_size)

model = KGCN\_model(n\_user, n\_entity, n\_relation, adj\_entity, adj\_relation, neighbor\_size, iter\_size=1, dim=16, l2=1e-7, aggregator='sum')

train(model, train\_data, test\_data, topk\_data, optimizer=tf.keras.optimizers.Adam(0.01), epochs=10, batch=512)

algorithm/KGCN/model.py

if \_\_name\_\_ == '\_\_main\_\_':

import Recommender\_System.utility.gpu\_memory\_growth

from Recommender\_System.algorithm.KGCN.tool import construct\_undirected\_kg, get\_adj\_list

from Recommender\_System.algorithm.KGCN.model import KGCN\_model

from Recommender\_System.algorithm.KGCN.train import train

from Recommender\_System.data import kg\_loader, data\_process

import tensorflow as tf

n\_user, n\_item, n\_entity, n\_relation, train\_data, test\_data, kg, topk\_data = data\_process.pack\_kg(kg\_loader.ml1m\_kg1m, negative\_sample\_threshold=4)

neighbor\_size = 16

adj\_entity, adj\_relation = get\_adj\_list(construct\_undirected\_kg(kg), n\_entity, neighbor\_size)

model = KGCN\_model(n\_user, n\_entity, n\_relation, adj\_entity, adj\_relation, neighbor\_size, iter\_size=1, dim=16, l2=1e-7, aggregator='sum')

train(model, train\_data, test\_data, topk\_data, optimizer=tf.keras.optimizers.Adam(0.01), epochs=10, batch=512)

algorithm/KGCN/train.py

import time

from typing import List, Tuple

import tensorflow as tf

from Recommender\_System.utility.decorator import logger

from Recommender\_System.utility.evaluation import TopkData

from Recommender\_System.algorithm.train import prepare\_ds, get\_score\_fn

from Recommender\_System.algorithm.common import log, topk

@logger('开始训练，', ('epochs', 'batch'))

def train(model: tf.keras.Model, train\_data: List[Tuple[int, int, int]], test\_data: List[Tuple[int, int, int]],

topk\_data: TopkData = None, optimizer=None, epochs=100, batch=512):

if optimizer is None:

optimizer = tf.keras.optimizers.Adam()

train\_ds, test\_ds = prepare\_ds(train\_data, test\_data, batch)

loss\_mean\_metric = tf.keras.metrics.Mean()

auc\_metric = tf.keras.metrics.AUC()

precision\_metric = tf.keras.metrics.Precision()

recall\_metric = tf.keras.metrics.Recall()

loss\_object = tf.keras.losses.BinaryCrossentropy()

if topk\_data:

score\_fn = get\_score\_fn(model)

def reset\_metrics():

for metric in [loss\_mean\_metric, auc\_metric, precision\_metric, recall\_metric]:

tf.py\_function(metric.reset\_states, [], [])

def update\_metrics(loss, label, score):

loss\_mean\_metric.update\_state(loss)

auc\_metric.update\_state(label, score)

precision\_metric.update\_state(label, score)

recall\_metric.update\_state(label, score)

def get\_metric\_results():

return loss\_mean\_metric.result(), auc\_metric.result(), precision\_metric.result(), recall\_metric.result()

@tf.function

def train\_batch(ui, label):

with tf.GradientTape() as tape:

score = model(ui, training=True)

loss = loss\_object(label, score) + sum(model.losses)

gradients = tape.gradient(loss, model.trainable\_variables)

optimizer.apply\_gradients(zip(gradients, model.trainable\_variables))

update\_metrics(loss, label, score)

@tf.function

def test\_batch(ui, label):

score = model(ui)

loss = loss\_object(label, score) + sum(model.losses)

update\_metrics(loss, label, score)

for epoch in range(epochs):

epoch\_start\_time = time.time()

reset\_metrics()

for ui, label in train\_ds:

train\_batch(ui, label)

train\_loss, train\_auc, train\_precision, train\_recall = get\_metric\_results()

reset\_metrics()

for ui, label in test\_ds:

test\_batch(ui, label)

test\_loss, test\_auc, test\_precision, test\_recall = get\_metric\_results()

log(epoch, train\_loss, train\_auc, train\_precision, train\_recall, test\_loss, test\_auc, test\_precision, test\_recall)

if topk\_data:

topk(topk\_data, score\_fn)

print('epoch\_time=', time.time() - epoch\_start\_time, 's', sep='')

algorithm/KGCN/tool.py

from Recommender\_System.utility.decorator import logger

from typing import List, Tuple, Dict

from collections import defaultdict

import numpy as np

@logger('根据知识图谱结构构建无向图')

def construct\_undirected\_kg(kg: List[Tuple[int, int, int]]) -> Dict[int, List[Tuple[int, int]]]:

kg\_dict = defaultdict(list)

for head\_id, relation\_id, tail\_id in kg:

kg\_dict[head\_id].append((relation\_id, tail\_id))

kg\_dict[tail\_id].append((relation\_id, head\_id)) # 将知识图谱视为无向图

return kg\_dict

@logger('根据知识图谱无向图构建邻接表，', ('n\_entity', 'neighbor\_size'))

def get\_adj\_list(kg\_dict: Dict[int, List[Tuple[int, int]]], n\_entity: int, neighbor\_size: int) ->\

Tuple[List[List[int]], List[List[int]]]:

adj\_entity, adj\_relation = [None for \_ in range(n\_entity)], [None for \_ in range(n\_entity)]

for entity\_id in range(n\_entity):

neighbors = kg\_dict[entity\_id]

n\_neighbor = len(neighbors)

sample\_indices = np.random.choice(range(n\_neighbor), size=neighbor\_size, replace=n\_neighbor < neighbor\_size)

adj\_relation[entity\_id] = [neighbors[i][0] for i in sample\_indices]

adj\_entity[entity\_id] = [neighbors[i][1] for i in sample\_indices]

return adj\_entity, adj\_relation

algorithm/MKR/layer.py

import tensorflow as tf

class CrossLayer(tf.keras.layers.Layer):

def call(self, inputs):

v, e = inputs # (batch, dim)

v = tf.expand\_dims(v, axis=2) # (batch, dim, 1)

e = tf.expand\_dims(e, axis=1) # (batch, 1, dim)

c\_matrix = tf.matmul(v, e) # (batch, dim, dim)

c\_matrix\_t = tf.transpose(c\_matrix, perm=[0, 2, 1]) # (batch, dim, dim)

return c\_matrix, c\_matrix\_t

class CompressLayer(tf.keras.layers.Layer):

def \_\_init\_\_(self, weight\_regularizer, \*\*kwargs):

super(CompressLayer, self).\_\_init\_\_(\*\*kwargs)

self.weight\_regularizer = tf.keras.regularizers.get(weight\_regularizer)

def build(self, input\_shape):

self.dim = input\_shape[0][-1]

self.weight = self.add\_weight(shape=(self.dim, 1), regularizer=self.weight\_regularizer, name='weight')

self.weight\_t = self.add\_weight(shape=(self.dim, 1), regularizer=self.weight\_regularizer, name='weight\_t')

self.bias = self.add\_weight(shape=self.dim, initializer='zeros', name='bias')

def call(self, inputs):

c\_matrix, c\_matrix\_t = inputs # (batch, dim, dim)

c\_matrix = tf.reshape(c\_matrix, shape=[-1, self.dim]) # (batch \* dim, dim)

c\_matrix\_t = tf.reshape(c\_matrix\_t, shape=[-1, self.dim]) # (batch \* dim, dim)

return tf.reshape(tf.matmul(c\_matrix, self.weight) + tf.matmul(c\_matrix\_t, self.weight\_t),

shape=[-1, self.dim]) + self.bias # (batch, dim)

def cross\_compress\_unit(inputs, weight\_regularizer):

cross\_feature\_matrix = CrossLayer()(inputs)

v\_out = CompressLayer(weight\_regularizer)(cross\_feature\_matrix)

e\_out = CompressLayer(weight\_regularizer)(cross\_feature\_matrix)

return v\_out, e\_out

algorithm/MKR/main.py

if \_\_name\_\_ == '\_\_main\_\_':

import Recommender\_System.utility.gpu\_memory\_growth

from tensorflow.keras.optimizers import Adam

from Recommender\_System.data import kg\_loader, data\_process

from Recommender\_System.algorithm.MKR.model import MKR\_model

from Recommender\_System.algorithm.MKR.train import train

n\_user, n\_item, n\_entity, n\_relation, train\_data, test\_data, kg, topk\_data = data\_process.pack\_kg(kg\_loader.ml1m\_kg20k, keep\_all\_head=False, negative\_sample\_threshold=4)

model\_rs, model\_kge = MKR\_model(n\_user, n\_item, n\_entity, n\_relation, dim=8, L=1, H=1, l2=1e-6)

train(model\_rs, model\_kge, train\_data, test\_data, kg, topk\_data, kge\_interval=3,

optimizer\_rs=Adam(0.02), optimizer\_kge=Adam(0.01), epochs=20, batch=4096)

'''

n\_user, n\_item, n\_entity, n\_relation, train\_data, test\_data, kg, topk\_data = data\_process.pack\_kg(kg\_loader.lastfm\_kg15k, keep\_all\_head=False)

model\_rs, model\_kge = MKR\_model(n\_user, n\_item, n\_entity, n\_relation, dim=4, L=2, H=1, l2=1e-6)

train(model\_rs, model\_kge, train\_data, test\_data, kg, topk\_data, kge\_interval=2,

optimizer\_rs=Adam(1e-3), optimizer\_kge=Adam(2e-4), epochs=10, batch=256)

'''

'''

n\_user, n\_item, n\_entity, n\_relation, train\_data, test\_data, kg, topk\_data = data\_process.pack\_kg(kg\_loader.bx\_kg20k, keep\_all\_head=False)

model\_rs, model\_kge = MKR\_model(n\_user, n\_item, n\_entity, n\_relation, dim=8, L=1, H=1, l2=1e-6)

train(model\_rs, model\_kge, train\_data, test\_data, kg, topk\_data, kge\_interval=2,

optimizer\_rs=Adam(2e-4), optimizer\_kge=Adam(2e-5), epochs=10, batch=32)

'''

Algorithm/MKR/model.py

from typing import Tuple

import tensorflow as tf

from Recommender\_System.algorithm.MKR.layer import cross\_compress\_unit

from Recommender\_System.utility.decorator import logger

@logger('初始化MKR模型：', ('n\_user', 'n\_item', 'n\_entity', 'n\_relation', 'dim', 'L', 'H', 'l2'))

def MKR\_model(n\_user: int, n\_item: int, n\_entity: int, n\_relation: int, dim=8, L=1, H=1, l2=1e-6) -> Tuple[tf.keras.Model, tf.keras.Model]:

l2 = tf.keras.regularizers.l2(l2)

user\_id = tf.keras.Input(shape=(), name='user\_id', dtype=tf.int32)

item\_id = tf.keras.Input(shape=(), name='item\_id', dtype=tf.int32)

head\_id = tf.keras.Input(shape=(), name='head\_id', dtype=tf.int32)

relation\_id = tf.keras.Input(shape=(), name='relation\_id', dtype=tf.int32)

tail\_id = tf.keras.Input(shape=(), name='tail\_id', dtype=tf.int32)

user\_embedding = tf.keras.layers.Embedding(n\_user, dim, embeddings\_regularizer=l2)

item\_embedding = tf.keras.layers.Embedding(n\_item, dim, embeddings\_regularizer=l2)

entity\_embedding = tf.keras.layers.Embedding(n\_entity, dim, embeddings\_regularizer=l2)

relation\_embedding = tf.keras.layers.Embedding(n\_relation, dim, embeddings\_regularizer=l2)

u = user\_embedding(user\_id)

i = item\_embedding(item\_id)

h = entity\_embedding(head\_id)

r = relation\_embedding(relation\_id)

t = entity\_embedding(tail\_id)

for \_ in range(L):

u = tf.keras.layers.Dense(dim, activation='relu', kernel\_regularizer=l2)(u)

i, h = cross\_compress\_unit(inputs=(i, h), weight\_regularizer=l2)

t = tf.keras.layers.Dense(dim, activation='relu', kernel\_regularizer=l2)(t)

#rs = tf.concat([u, i], axis=1)

rs = tf.keras.activations.sigmoid(tf.reduce\_sum(u \* i, axis=1, keepdims=True))

kge = tf.concat([h, r], axis=1)

for \_ in range(H - 1):

#rs = tf.keras.layers.Dense(dim \* 2, activation='relu', kernel\_regularizer=reg\_l2(l2))(rs)

kge = tf.keras.layers.Dense(dim \* 2, activation='relu', kernel\_regularizer=l2)(kge)

#rs = tf.keras.layers.Dense(1, activation='sigmoid', kernel\_regularizer=reg\_l2(l2))(rs)

kge = tf.keras.layers.Dense(dim, activation='sigmoid', kernel\_regularizer=l2)(kge)

kge = -tf.keras.activations.sigmoid(tf.reduce\_sum(t \* kge, axis=1))

return tf.keras.Model(inputs=[user\_id, item\_id, head\_id], outputs=rs),\

tf.keras.Model(inputs=[item\_id, head\_id, relation\_id, tail\_id], outputs=kge)

if \_\_name\_\_ == '\_\_main\_\_':

rs\_model, kge\_model = MKR\_model(2, 2, 2, 2)

u = tf.constant([0, 1])

i = tf.constant([1, 0])

h = tf.constant([0, 1])

r = tf.constant([1, 0])

t = tf.constant([0, 1])

print(rs\_model({'user\_id': u, 'item\_id': i, 'head\_id': h}))

print(kge\_model({'item\_id': i, 'head\_id': h, 'relation\_id': r, 'tail\_id': t}))

ds = tf.data.Dataset.from\_tensor\_slices(({'item\_id': i, 'head\_id': h, 'relation\_id': r, 'tail\_id': t}, tf.constant([0] \* 2))).batch(2)

kge\_model.compile(optimizer='adam', loss=lambda y\_true, y\_pre: y\_pre)

kge\_model.fit(ds, epochs=3)

#ds = tf.data.Dataset.from\_tensor\_slices(({'user\_id': u, 'item\_id': i, 'head\_id': h}, tf.constant([0., 1.]))).batch(2)

#rs\_model.compile(optimizer='adam', loss=tf.keras.losses.BinaryCrossentropy())

#rs\_model.fit(ds, epochs=3)

algorithm/MKR/train.py

from typing import List, Tuple

import tensorflow as tf

from Recommender\_System.algorithm.train import RsCallback

from Recommender\_System.utility.evaluation import TopkData

from Recommender\_System.utility.decorator import logger

class \_KgeCallback(tf.keras.callbacks.Callback):

def on\_epoch\_end(self, epoch, logs=None):

tf.print('KGE: epoch=', epoch + 1, ', loss=', logs['loss'], sep='')

def \_get\_score\_fn(model):

@tf.function(experimental\_relax\_shapes=True)

def \_fast\_model(inputs):

return tf.squeeze(model(inputs))

def \_score\_fn(inputs):

inputs = {k: tf.constant(v, dtype=tf.int32) for k, v in inputs.items()}

inputs['head\_id'] = inputs['item\_id']

return \_fast\_model(inputs).numpy()

return \_score\_fn

@logger('开始训练，', ('epochs', 'batch'))

def train(model\_rs: tf.keras.Model, model\_kge: tf.keras.Model, train\_data: List[Tuple[int, int, int]],

test\_data: List[Tuple[int, int, int]], kg: List[Tuple[int, int, int]], topk\_data: TopkData,

optimizer\_rs=None, optimizer\_kge=None, kge\_interval=3, epochs=100, batch=512):

if optimizer\_rs is None:

optimizer\_rs = tf.keras.optimizers.Adam()

if optimizer\_kge is None:

optimizer\_kge = tf.keras.optimizers.Adam()

def xy(data):

user\_id = tf.constant([d[0] for d in data], dtype=tf.int32)

item\_id = tf.constant([d[1] for d in data], dtype=tf.int32)

head\_id = tf.constant([d[1] for d in data], dtype=tf.int32)

label = tf.constant([d[2] for d in data], dtype=tf.float32)

return {'user\_id': user\_id, 'item\_id': item\_id, 'head\_id': head\_id}, label

def xy\_kg(kg):

item\_id = tf.constant([d[0] for d in kg], dtype=tf.int32)

head\_id = tf.constant([d[0] for d in kg], dtype=tf.int32)

relation\_id = tf.constant([d[1] for d in kg], dtype=tf.int32)

tail\_id = tf.constant([d[2] for d in kg], dtype=tf.int32)

label = tf.constant([0] \* len(kg), dtype=tf.float32)

return {'item\_id': item\_id, 'head\_id': head\_id, 'relation\_id': relation\_id, 'tail\_id': tail\_id}, label

train\_ds = tf.data.Dataset.from\_tensor\_slices(xy(train\_data)).shuffle(len(train\_data)).batch(batch)

test\_ds = tf.data.Dataset.from\_tensor\_slices(xy(test\_data)).batch(batch)

kg\_ds = tf.data.Dataset.from\_tensor\_slices(xy\_kg(kg)).shuffle(len(kg)).batch(batch)

model\_rs.compile(optimizer=optimizer\_rs, loss='binary\_crossentropy', metrics=['AUC', 'Precision', 'Recall'])

model\_kge.compile(optimizer=optimizer\_kge, loss=lambda y\_true, y\_pre: y\_pre)

for epoch in range(epochs):

model\_rs.fit(train\_ds, epochs=epoch + 1, verbose=0, validation\_data=test\_ds,

callbacks=[RsCallback(topk\_data, \_get\_score\_fn(model\_rs))], initial\_epoch=epoch)

if epoch % kge\_interval == 0:

model\_kge.fit(kg\_ds, epochs=epoch + 1, verbose=0, callbacks=[\_KgeCallback()], initial\_epoch=epoch)

algorithm/common.py

from typing import List, Callable, Dict

from Recommender\_System.utility.evaluation import TopkData, topk\_evaluate

def log(epoch, train\_loss, train\_auc, train\_precision, train\_recall, test\_loss, test\_auc, test\_precision, test\_recall):

train\_f1 = 2. \* train\_precision \* train\_recall / pr if (pr := train\_precision + train\_recall) else 0

test\_f1 = 2. \* test\_precision \* test\_recall / pr if (pr := test\_precision + test\_recall) else 0

print('epoch=%d, train\_loss=%.5f, train\_auc=%.5f, train\_f1=%.5f, test\_loss=%.5f, test\_auc=%.5f, test\_f1=%.5f' %

(epoch + 1, train\_loss, train\_auc, train\_f1, test\_loss, test\_auc, test\_f1))

def topk(topk\_data: TopkData, score\_fn: Callable[[Dict[str, List[int]]], List[float]], ks=[10, 36, 100]):

precisions, recalls = topk\_evaluate(topk\_data, score\_fn, ks)

for k, precision, recall in zip(ks, precisions, recalls):

f1 = 2. \* precision \* recall / pr if (pr := precision + recall) else 0

print('[k=%d, precision=%.3f%%, recall=%.3f%%, f1=%.3f%%]' %

(k, 100. \* precision, 100. \* recall, 100. \* f1), end='')

print()

algorithm/train.py (not-a-script meaning a dependency file)

from typing import List, Tuple, Callable, Dict

import tensorflow as tf

from Recommender\_System.algorithm.common import log, topk

from Recommender\_System.utility.evaluation import TopkData

from Recommender\_System.utility.decorator import logger

def prepare\_ds(train\_data: List[Tuple[int, int, int]], test\_data: List[Tuple[int, int, int]],

batch: int) -> Tuple[tf.data.Dataset, tf.data.Dataset]:

def xy(data):

user\_ids = tf.constant([d[0] for d in data], dtype=tf.int32)

item\_ids = tf.constant([d[1] for d in data], dtype=tf.int32)

labels = tf.constant([d[2] for d in data], dtype=tf.keras.backend.floatx())

return {'user\_id': user\_ids, 'item\_id': item\_ids}, labels

train\_ds = tf.data.Dataset.from\_tensor\_slices(xy(train\_data)).shuffle(len(train\_data)).batch(batch)

test\_ds = tf.data.Dataset.from\_tensor\_slices(xy(test\_data)).batch(batch)

return train\_ds, test\_ds

def \_evaluate(model, dataset, loss\_object, mean\_metric=tf.keras.metrics.Mean(), auc\_metric=tf.keras.metrics.AUC(),

precision\_metric=tf.keras.metrics.Precision(), recall\_metric=tf.keras.metrics.Recall()):

for metric in [mean\_metric, auc\_metric, precision\_metric, recall\_metric]:

tf.py\_function(metric.reset\_states, [], [])

@tf.function

def evaluate\_batch(ui, label):

score = tf.squeeze(model(ui))

loss = loss\_object(label, score) + sum(model.losses)

return score, loss

for ui, label in dataset:

score, loss = evaluate\_batch(ui, label)

mean\_metric.update\_state(loss)

auc\_metric.update\_state(label, score)

precision\_metric.update\_state(label, score)

recall\_metric.update\_state(label, score)

return mean\_metric.result(), auc\_metric.result(), precision\_metric.result(), recall\_metric.result()

def \_train\_graph(model, train\_ds, test\_ds, topk\_data, optimizer, loss\_object, epochs):

score\_fn = get\_score\_fn(model)

@tf.function

def train\_batch(ui, label):

with tf.GradientTape() as tape:

score = tf.squeeze(model(ui, training=True))

loss = loss\_object(label, score) + sum(model.losses)

gradients = tape.gradient(loss, model.trainable\_variables)

optimizer.apply\_gradients(zip(gradients, model.trainable\_variables))

for epoch in range(epochs):

for ui, label in train\_ds:

train\_batch(ui, label)

train\_loss, train\_auc, train\_precision, train\_recall = \_evaluate(model, train\_ds, loss\_object)

test\_loss, test\_auc, test\_precision, test\_recall = \_evaluate(model, test\_ds, loss\_object)

log(epoch, train\_loss, train\_auc, train\_precision, train\_recall, test\_loss, test\_auc, test\_precision, test\_recall)

topk(topk\_data, score\_fn)

def \_train\_eager(model, train\_ds, test\_ds, topk\_data, optimizer, loss\_object, epochs):

model.compile(optimizer=optimizer, loss=loss\_object, metrics=['AUC', 'Precision', 'Recall'])

model.fit(train\_ds, epochs=epochs, verbose=0, validation\_data=test\_ds,

callbacks=[RsCallback(topk\_data, get\_score\_fn(model))])

class RsCallback(tf.keras.callbacks.Callback):

def \_\_init\_\_(self, topk\_data: TopkData, score\_fn: Callable[[Dict[str, List[int]]], List[float]]):

super(RsCallback, self).\_\_init\_\_()

self.topk\_data = topk\_data

self.score\_fn = score\_fn

def on\_epoch\_end(self, epoch, logs=None):

log(epoch, logs['loss'], logs['auc'], logs['precision'], logs['recall'],

logs['val\_loss'], logs['val\_auc'], logs['val\_precision'], logs['val\_recall'])

topk(self.topk\_data, self.score\_fn)

@logger('开始训练，', ('epochs', 'batch', 'execution'))

def train(model: tf.keras.Model, train\_data: List[Tuple[int, int, int]], test\_data: List[Tuple[int, int, int]],

topk\_data: TopkData, optimizer=None, loss\_object=None, epochs=100, batch=512, execution='eager') -> None:

"""

通用训练流程。

:param model: 模型

:param train\_data: 训练集

:param test\_data: 测试集

:param topk\_data: 用于topk评估数据

:param optimizer: 优化器，默认为Adam

:param loss\_object: 损失函数，默认为BinaryCrossentropy

:param epochs: 迭代次数

:param batch: 批数量

:param execution: 执行模式，为eager或graph。在eager模式下，用model.fit；在graph模式下，用tf.function和GradientTape

"""

if optimizer is None:

optimizer = tf.keras.optimizers.Adam()

if loss\_object is None:

loss\_object = tf.keras.losses.BinaryCrossentropy()

train\_ds, test\_ds = prepare\_ds(train\_data, test\_data, batch)

train\_fn = \_train\_eager if execution == 'eager' else \_train\_graph

train\_fn(model, train\_ds, test\_ds, topk\_data, optimizer, loss\_object, epochs)

@logger('开始测试，', ('batch',))

def test(model: tf.keras.Model, train\_data: List[Tuple[int, int, int]], test\_data: List[Tuple[int, int, int]],

topk\_data: TopkData, loss\_object=None, batch=512) -> None:

"""

通用测试流程。

:param model: 模型

:param train\_data: 训练集

:param test\_data: 测试集

:param topk\_data: 用于topk评估数据

:param loss\_object: 损失函数，默认为BinaryCrossentropy

:param batch: 批数量

"""

if loss\_object is None:

loss\_object = tf.keras.losses.BinaryCrossentropy()

train\_ds, test\_ds = prepare\_ds(train\_data, test\_data, batch)

train\_loss, train\_auc, train\_precision, train\_recall = \_evaluate(model, train\_ds, loss\_object)

test\_loss, test\_auc, test\_precision, test\_recall = \_evaluate(model, test\_ds, loss\_object)

log(-1, train\_loss, train\_auc, train\_precision, train\_recall, test\_loss, test\_auc, test\_precision, test\_recall)

topk(topk\_data, get\_score\_fn(model))

def get\_score\_fn(model):

@tf.function(experimental\_relax\_shapes=True)

def \_fast\_model(ui):

return tf.squeeze(model(ui))

def score\_fn(ui):

ui = {k: tf.constant(v, dtype=tf.int32) for k, v in ui.items()}

return \_fast\_model(ui).numpy()

return score\_fn