

---

# **DeepRec**

***Release latest***

**Alibaba Group Holding Limited**

**Sep 20, 2023**



# BUILD INSTALL

<b>1</b>	<b>Background</b>	<b>3</b>
<b>2</b>	<b>Getting started</b>	<b>5</b>
2.1	DeepRec Build and Install . . . . .	5
2.2	Estimator Build and Install . . . . .	7
2.3	TFServing Build and Install . . . . .	8
<b>3</b>	<b>Features</b>	<b>11</b>
3.1	Embedding Variable . . . . .	11
3.2	Feature Eviction of EmbeddingVariable . . . . .	17
3.3	Feature Filter of EmbeddingVariable . . . . .	19
3.4	Adaptive Embedding . . . . .	23
3.5	Multi-Hash Variable . . . . .	24
3.6	Group Embedding . . . . .	29
3.7	GRPC++ . . . . .	35
3.8	StarServer . . . . .	37
3.9	Distributed Synchronous Training-SOK . . . . .	39
3.10	Embedding Subgraph Fusion . . . . .	43
3.11	Pipeline-Stage . . . . .	48
3.12	Pipeline-SmartStage . . . . .	50
3.13	Asynchronous Embedding Lookup . . . . .	55
3.14	Sample-awared Graph Compression . . . . .	57
3.15	CPU Memory Optimization . . . . .	60
3.16	GPU Memory Optimization . . . . .	61
3.17	GPU Virtual Memory Optimization . . . . .	62
3.18	Executor Optimization . . . . .	62
3.19	GPU Multi-stream . . . . .	63
3.20	Incremental Checkpoint . . . . .	66
3.21	Embedding Variable Export Format . . . . .	68
3.22	AdamAsync Optimizer . . . . .	71
3.23	AdagradDecay Optimizer . . . . .	73
3.24	AdamW Optimizer . . . . .	75
3.25	oneDNN . . . . .	77
3.26	Optimization of Operator . . . . .	78
3.27	NVIDIA TF32 . . . . .	81
3.28	Arm Compute Library . . . . .	83
3.29	Intel® AMX . . . . .	84
3.30	BFloat16 . . . . .	86
3.31	WorkQueue . . . . .	89
3.32	KafkaDataset . . . . .	91

3.33	KafkaGroupIODataset . . . . .	92
3.34	ParquetDataset . . . . .	93
3.35	TensorRT . . . . .	97
3.36	BladeDISC . . . . .	99
3.37	XLA . . . . .	102
3.38	Processor . . . . .	103
3.39	SessionGroup . . . . .	114
3.40	Device Placement Optimization . . . . .	123

DeepRec is a recommendation engine based on TensorFlow 1.15, Intel-TensorFlow and NVIDIA-TensorFlow.



## **BACKGROUND**

Sparse model is a type of deep learning model that accounts for a relatively high proportion of discrete feature calculation logic in the model structure. Discrete features are usually expressed as non-numeric features that cannot be directly processed by algorithms such as id, tag, text, and phrases. They are widely used in high-value businesses such as search, advertising, and recommendation.

DeepRec has been deeply cultivated since 2016, which supports core businesses such as Taobao Search, recommendation and advertising. It precipitates a list of features on basic frameworks and has excellent performance in sparse models training. Facing a wide variety of external needs and the environment of deep learning framework embracing open source, DeepRec open source is conducive to establishing standardized interfaces, cultivating user habits, greatly reducing the cost of external customers working on cloud and establishing the brand value.





## GETTING STARTED

## 2.1 DeepRec Build and Install

### 2.1.1 Setup

#### CPU Base Docker Image

GCC Version	Python Version	IMAGE
7.5.0	3.6.9	alideeprec/deeprec-base:deeprec-base-cpu-py36-ubuntu18.04
9.4.0	3.8.10	alideeprec/deeprec-base:deeprec-base-cpu-py38-ubuntu20.04
11.2.0	3.8.6	alideeprec/deeprec-base:deeprec-base-cpu-py38-ubuntu22.04

#### GPU Base Docker Image

GCC Version	Python Version	CUDA VERSION	IMAGE
7.5.0	3.6.9	CUDA 11.6.1	alideeprec/deeprec-base:deeprec-base-gpu-py36-cu116-ubuntu18.04
9.4.0	3.8.10	CUDA 11.6.2	alideeprec/deeprec-base:deeprec-base-gpu-py38-cu116-ubuntu20.04
11.2.0	3.8.6	CUDA 11.7.1	alideeprec/deeprec-base:deeprec-base-gpu-py38-cu117-ubuntu22.04

#### CPU Dev Docker (with bazel cache)

GCC Version	Python Version	IMAGE
7.5.0	3.6.9	alideeprec/deeprec-build:deeprec-dev-cpu-py36-ubuntu18.04
9.4.0	3.8.10	alideeprec/deeprec-build:deeprec-dev-cpu-py38-ubuntu20.04

#### GPU(cuda11.6) Dev Docker (with bazel cache)

GCC Version	Python Version	CUDA VERSION	IMAGE
7.5.0	3.6.9	CUDA 11.6.1	alideeprec/deeprec-build:deeprec-dev-gpu-py36-cu116-ubuntu18.04
9.4.0	3.8.10	CUDA 11.6.2	alideeprec/deeprec-build:deeprec-dev-gpu-py38-cu116-ubuntu20.04

## 2.1.2 Build

### GPU Environment

Configure TF\_CUDA\_COMPUTE\_CAPABILITIES could improve performance, please follow to setup correct TF\_CUDA\_COMPUTE\_CAPABILITIES.

GPU architecture	TF_CUDA_COMPUTE_CAPABILITIES
Pascal (P100)	6.0+6.1
Volta (V100)	7.0
Turing (T4)	7.5
Ampere (A10, A100)	8.0+8.6
Hopper (H100, H800)	9.0

If you need to compile DeepRec wheel that run on different GPU architecture, configure TF\_CUDA\_COMPUTE\_CAPABILITIES, by default TF\_CUDA\_COMPUTE\_CAPABILITIES is “7.0,7.5,8.0,8.6” (CIBUILD use A10 card).

For example, if you want to run DeepRec on H100 and A100 GPU card, please setup TF\_CUDA\_COMPUTE\_CAPABILITIES as follows:

```
export TF_CUDA_COMPUTE_CAPABILITIES="8.0,8.6,9.0"
```

### Configuration

```
./configure
```

### Build GPU/CPU Package Builder

```
bazel build -c opt --config=opt //tensorflow/tools/pip_package:build_pip_package
```

### Build GPU/CPU Package Builder with ABI=0

```
bazel build --cxxopt="-D_GLIBCXX_USE_CXX11_ABI=0" --host_cxxopt="-D_GLIBCXX_USE_CXX11_
↪ABI=0" -c opt --config=opt //tensorflow/tools/pip_package:build_pip_package
```

### Build CPU Package Builder with OneDNN + Eigen Threadpool

```
bazel build -c opt --config=opt --config=mkl_threadpool --define build_with_mkl_dnn_v1_
↪only=true //tensorflow/tools/pip_package:build_pip_package
```

### Build CPU Package Builder with OneDNN + Eigen Threadpool + ABI=0

```
bazel build --cxxopt="-D_GLIBCXX_USE_CXX11_ABI=0" --host_cxxopt="-D_GLIBCXX_USE_CXX11_
↪ABI=0" -c opt --config=opt --config=mkl_threadpool --define build_with_mkl_dnn_v1_
↪only=true //tensorflow/tools/pip_package:build_pip_package
```

### Build ARM CPU package Builder with Arm Compute Library (ACL)

```
bazel build -c opt --config=opt --config=mkl_aarch64 //tensorflow/tools/pip_
↪package:build_pip_package
```

### 2.1.3 Build Package

```
./bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg
```

### 2.1.4 Install Package

```
pip3 install /tmp/tensorflow_pkg/tensorflow-1.15.5+${version}-cp38-cp38m-linux_x86_64.whl
```

### 2.1.5 Latest Release Images

#### CPU Image

x86\_64:

```
alideeprec/deeprec-release:deeprec2306-cpu-py38-ubuntu20.04
```

arm64:

```
alideeprec/deeprec-release:deeprec2302-cpu-py38-ubuntu22.04-arm64
```

#### GPU Image with CUDA 11.6

```
alideeprec/deeprec-release:deeprec2306-gpu-py38-cu116-ubuntu20.04
```

## 2.2 Estimator Build and Install

### 2.2.1 Setup

#### CPU Base Docker Image

GCC Version	Python Version	IMAGE
7.5.0	3.6.9	alideeprec/deeprec-base:deeprec-base-cpu-py36-ubuntu18.04
9.4.0	3.8.10	alideeprec/deeprec-base:deeprec-base-cpu-py38-ubuntu20.04
11.2.0	3.8.6	alideeprec/deeprec-base:deeprec-base-cpu-py38-ubuntu22.04

#### GPU Base Docker Image

GCC Version	Python Version	CUDA VERSION	IMAGE
7.5.0	3.6.9	CUDA 11.6.1	alideeprec/deeprec-base:deeprec-base-gpu-py36-cu116-ubuntu18.04
9.4.0	3.8.10	CUDA 11.6.2	alideeprec/deeprec-base:deeprec-base-gpu-py38-cu116-ubuntu20.04
11.2.0	3.8.6	CUDA 11.7.1	alideeprec/deeprec-base:deeprec-base-gpu-py38-cu117-ubuntu22.04

#### CPU Dev Docker (with bazel cache)

GCC Version	Python Version	IMAGE
7.5.0	3.6.9	alideeprec/deeprec-build:deeprec-dev-cpu-py36-ubuntu18.04
9.4.0	3.8.10	alideeprec/deeprec-build:deeprec-dev-cpu-py38-ubuntu20.04

**GPU(cuda11.6) Dev Docker (with bazel cache)**

GCC Version	Python Version	CUDA VERSION	IMAGE
7.5.0	3.6.9	CUDA 11.6.1	alideeprec/deeprec-build:deeprec-dev-gpu-py36-cu116-ubuntu18.04
9.4.0	3.8.10	CUDA 11.6.2	alideeprec/deeprec-build:deeprec-dev-gpu-py38-cu116-ubuntu20.04

## 2.2.2 Estimator Source Code

DeepRec provide new distributed protocols such as `grpc++` and `star_server`, which is not supported when use native Estimator. We provide Estimator based on DeepRec.

Source Code: <https://github.com/DeepRec-AI/estimator>

Develop Branch: `master`, Latest Release Branch: `deeprec2306`

## 2.2.3 Estimator Build

**Build Package Builder**

```
bazel build //tensorflow_estimator/tools/pip_package:build_pip_package
```

**Build Package**

```
bazel-bin/tensorflow_estimator/tools/pip_package/build_pip_package /tmp/estimator_whl
```

## 2.2.4 Estimator Install Package

Installing DeepRec will install the native tensorflow-estimator by default, please reinstall the compiled Estimator.

## 2.3 TFServing Build and Install

### 2.3.1 Setup

**CPU Base Docker Image**

GCC Version	Python Version	IMAGE
7.5.0	3.6.9	alideeprec/deeprec-base:deeprec-base-cpu-py36-ubuntu18.04
9.4.0	3.8.10	alideeprec/deeprec-base:deeprec-base-cpu-py38-ubuntu20.04
11.2.0	3.8.6	alideeprec/deeprec-base:deeprec-base-cpu-py38-ubuntu22.04

**GPU Base Docker Image**

GCC Version	Python Version	CUDA VERSION	IMAGE
7.5.0	3.6.9	CUDA 11.6.1	alideeprec/deeprec-base:deeprec-base-gpu-py36-cu116-ubuntu18.04
9.4.0	3.8.10	CUDA 11.6.2	alideeprec/deeprec-base:deeprec-base-gpu-py38-cu116-ubuntu20.04
11.2.0	3.8.6	CUDA 11.7.1	alideeprec/deeprec-base:deeprec-base-gpu-py38-cu117-ubuntu22.04

**CPU Dev Docker (with bazel cache)**

GCC Version	Python Version	IMAGE
7.5.0	3.6.9	alideeprec/deeprec-build:deeprec-dev-cpu-py36-ubuntu18.04
9.4.0	3.8.10	alideeprec/deeprec-build:deeprec-dev-cpu-py38-ubuntu20.04

**GPU(cuda11.6) Dev Docker (with bazel cache)**

GCC Version	Python Version	CUDA VERSION	IMAGE
7.5.0	3.6.9	CUDA 11.6.1	alideeprec/deeprec-build:deeprec-dev-gpu-py36-cu116-ubuntu18.04
9.4.0	3.8.10	CUDA 11.6.2	alideeprec/deeprec-build:deeprec-dev-gpu-py38-cu116-ubuntu20.04

## 2.3.2 TFServing Source Code

We provide optimized TFServing which could highly improve performance in inference, such as SessionGroup, CUDA multi-stream, etc.

Source Code: <https://github.com/DeepRec-AI/serving>

Develop Branch: master, Latest Release Branch: deeprec2306

## 2.3.3 TFServing Build

**Build Package Builder-CPU**

```
bazel build -c opt tensorflow_serving/...
```

**Build CPU Package Builder with OneDNN + Eigen Threadpool**

```
bazel build -c opt --config=mkl_threadpool --define build_with_mkl_dnn_v1_only=true ↵
↵ tensorflow_serving/...
```

**Build Package Builder-GPU**

```
bazel build -c opt --config=cuda tensorflow_serving/...
```

**Build Package**

```
bazel-bin/tensorflow_serving/tools/pip_package/build_pip_package /tmp/tf_serving_client_ ↵
↵ whl
```

**Server Bin**

Server Bin would generated in following directory:

`bazel-bin/tensorflow_serving/model_servers/tensorflow_model_server`

## **FEATURES**

### **3.1 Embedding Variable**

#### **3.1.1 Introduction**

Embedding parameters are saved in the form of `tf.Variable` in TensorFlow, and the size of `Variable` is `[vocabulary_size, embedding_dimension]` which needed to be decided before training and inference. This will bring difficulties for users in large-scale scenarios:

1. `vocabulary_size` is decided by the number of features, which makes it difficult to estimate `vocabulary_size` in online learning because of new features that keep coming.
2. The data type of the ids is string and their size are very large generally, so users need to hash the ids into values between 0 to `vocabulary_size` before lookup the embeddings:
  - The probability of different ids being mapped to the same embedding will increase significantly when the `vocabulary_size` is too small.;
  - Lots of memory will be wasted when the `vocabulary_size` is too big;
3. The enormous size of Embedding tables is the main reason for the increase of the model size. Even if the embedding of some features has little effect on the model through regularization, these embeddings cannot be removed from the model.

To solve the problems mentioned above, DeepRec provides `EmbeddingVariable` to support `Variable` with dynamic shape. With `EmbeddingVariable`, users can use memory efficiently while don't affect the accuracy of models, and making it easier to deploy the large-scale models.

`EmbeddingVariable` has undergone several iterations and currently support feature filter, feature eviction, feature statistics and other fundamental features. Moreover, optimizations including optimization on structure of sparse feature, lockless hash map, multi-tier embedding storage, Embedding GPU Ops, and GPU HashTable are added to `EmbeddingVariable`. Some features of `EmbeddingVariable` are also supported in TensorFlow recommenders-addons (<https://github.com/tensorflow/recommenders-addons/pull/16>).

### 3.1.2 User APIs

DeepRec provides users with three ways to create EmbeddingVariable:

#### Create EmbeddingVariable with `get_embedding_variable` API

```
def get_embedding_variable(name,
                           embedding_dim,
                           key_dtype=dtypes.int64,
                           value_dtype=None,
                           initializer=None,
                           trainable=True,
                           collections=None,
                           partitioner=None,
                           custom_getter=None,
                           ev_option = tf.EmbeddingVariableOption()):
```

- `name`: name of EmbeddingVariable
- `embedding_dim`: the dim of each embedding.
- `key_dtype`: data type of the key used to lookup embedding, default is int64, allowed values are int64 and int32
- `value_dtype`: the data type of embedding parameters, currently limited to float
- `initializer`: initial value of embedding parameters, initializer and list can be passed in
- `trainable`: whether to be added to the GraphKeys.TRAINABLE\_VARIABLES collection
- `collections`: list of graph collections keys to add the Variable to. Defaults to [GraphKeys.GLOBAL\_VARIABLES]
- `partitioner`: optional callable that accepts a fully defined TensorShape and dtype of the Variable to be created, and returns a list of partitions for each axis (currently only one axis can be partitioned)
- `custom_getter`: callable that takes as a first argument the true getter, and allows overwriting the internal `get_variable` method. The signature of `custom_getter` should match that of this method, but the most future-proof version will allow for changes: `def custom_getter(getter, *args, **kwargs)`. Direct access to all `get_variable` parameters is also allowed: `def custom_getter(getter, name, *args, **kwargs)`. A simple identity custom getter that simply creates variables with modified names is:

```
def custom_getter(getter, name, *args, **kwargs):
    return getter(name + '_suffix', *args, **kwargs)
```

- `ev_option`: options of EmbeddingVariable, e.g. options of feature filter and options of multi-tier storage

#### Create EmbeddingVariable with `tf.feature_column` API

```
def categorical_column_with_embedding(key,
                                     dtype=dtypes.string,
                                     partition_num=None,
                                     ev_option=tf.EmbeddingVariableOption()
                                     )
```

#### Create EmbeddingVariable with `tf.contrib.feature_column` API

```
def sparse_column_with_embedding(column_name,
                                 dtype=dtypes.string,
                                 partition_num=None,
```

(continues on next page)



(continued from previous page)

```

steps_to_live=None,
init_data_source=None,
ht_partition_num=1000,
evconfig = variables.EmbeddingVariableOption()

```

### 3.1.3 Demo

With `get_embedding_variable` API

```

import tensorflow as tf

var = tf.get_embedding_variable("var_0",
                               embedding_dim=3,
                               initializer=tf.ones_initializer(tf.float32),
                               partitioner=tf.fixed_size_partitioner(num_shards=4))

shape = [var1.total_count() for var1 in var]

emb = tf.nn.embedding_lookup(var, tf.cast([0,1,2,5,6,7], tf.int64))
fun = tf.multiply(emb, 2.0, name='multiply')
loss = tf.reduce_sum(fun, name='reduce_sum')
opt = tf.train.AdagradOptimizer(0.1)

g_v = opt.compute_gradients(loss)
train_op = opt.apply_gradients(g_v)

init = tf.global_variables_initializer()

sess_config = tf.ConfigProto(allow_soft_placement=True, log_device_placement=False)
with tf.Session(config=sess_config) as sess:
    sess.run([init])
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
    print(sess.run([shape]))

```

With `categorical_column_with_embedding` API

```

import tensorflow as tf
from tensorflow.python.framework import ops

columns = tf.feature_column.categorical_column_with_embedding("col_emb", dtype=tf.dtypes.
    ↪int64)
W = tf.feature_column.embedding_column(categorical_column=columns,
    dimension=3,
    initializer=tf.ones_initializer(tf.dtypes.float32))

ids={}
ids["col_emb"] = tf.SparseTensor(indices=[[0,0],[1,1],[2,2],[3,3],[4,4]], values=tf.
    ↪cast([1,2,3,4,5], tf.dtypes.int64), dense_shape=[5, 5])

```

(continues on next page)

(continued from previous page)

```

emb = tf.feature_column.input_layer(ids, [W])
fun = tf.multiply(emb, 2.0, name='multiply')
loss = tf.reduce_sum(fun, name='reduce_sum')
opt = tf.train.FtrlOptimizer(0.1, l1_regularization_strength=2.0, l2_regularization_
↪strength=0.00001)
g_v = opt.compute_gradients(loss)
train_op = opt.apply_gradients(g_v)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    print("init global done")
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))

```

**With sparse\_column\_with\_embedding API**

```

import tensorflow as tf
from tensorflow.python.framework import ops
from tensorflow.contrib.layers.python.layers import feature_column_ops
from tensorflow.contrib.layers.python.layers import feature_column

columns = feature_column.sparse_column_with_embedding(column_name="col_emb", dtype=tf.
↪dtypes.int64)
W = feature_column.embedding_column(sparse_id_column=columns,
    dimension=3,
    initializer=tf.ones_initializer(tf.dtypes.float32))

ids={}
ids["col_emb"] = tf.SparseTensor(indices=[[0,0],[1,1],[2,2],[3,3],[4,4]], values=tf.
↪cast([1,2,3,4,5], tf.dtypes.int64), dense_shape=[5, 5])

emb = feature_column_ops.input_from_feature_columns(columns_to_tensors=ids, feature_
↪columns=[W])
fun = tf.multiply(emb, 2.0, name='multiply')
loss = tf.reduce_sum(fun, name='reduce_sum')
opt = tf.train.FtrlOptimizer(0.1, l1_regularization_strength=2.0, l2_regularization_
↪strength=0.00001)
g_v = opt.compute_gradients(loss)
train_op = opt.apply_gradients(g_v)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    print("init global done")
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))

```

**With sequence\_categorical\_column\_with\_embedding API**

```

import tensorflow as tf
from tensorflow.python.feature_column import sequence_feature_column

columns = sequence_feature_column.sequence_categorical_column_with_embedding(key="col_emb",
↳ dtype=tf.dtypes.int32)
W = tf.feature_column.embedding_column(categorical_column=columns,
    dimension=3,
    initializer=tf.ones_initializer(tf.dtypes.float32))

ids={}
ids["col_emb"] = tf.SparseTensor(indices=[[0,0],[0,1],[1,1],[2,2],[3,3],[4,4]], \
    values=tf.cast([1,3,2,3,4,5], tf.dtypes.int64),
    dense_shape=[5, 5])

emb, length = tf.contrib.feature_column.sequence_input_layer(ids, [W])
fun = tf.multiply(emb, 2.0, name='multiply')
loss = tf.reduce_sum(fun, name='reduce_sum')
opt = tf.train.FtrlOptimizer(0.1, l1_regularization_strength=2.0, l2_regularization_
↳ strength=0.00001)
g_v = opt.compute_gradients(loss)
train_op = opt.apply_gradients(g_v)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    print("init global done")
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))

```

#### With weighted\_categorical\_column API

```

import tensorflow as tf

categorical_column = tf.feature_column.categorical_column_with_embedding("col_emb",
↳ dtype=tf.dtypes.int64)

ids={}
ids["col_emb"] = tf.SparseTensor(indices=[[0,0],[0,1],[1,1],[2,2],[3,3],[4,3],[4,4]], \
    values=tf.cast([1,3,2,3,4,5,3], tf.dtypes.int64), dense_shape=[5,
↳ 5])
ids['weight'] = [[2.0],[5.0],[4.0],[8.0],[3.0],[1.0],[2.5]]

columns = tf.feature_column.weighted_categorical_column(categorical_column, 'weight')

W = tf.feature_column.embedding_column(categorical_column=columns,
    dimension=3,
    initializer=tf.ones_initializer(tf.dtypes.float32))
emb = tf.feature_column.input_layer(ids, [W])
fun = tf.multiply(emb, 2.0, name='multiply')
loss = tf.reduce_sum(fun, name='reduce_sum')

```

(continues on next page)

(continued from previous page)

```

opt = tf.train.FtrlOptimizer(0.1, l1_regularization_strength=2.0, l2_regularization_
↪strength=0.00001)
g_v = opt.compute_gradients(loss)
train_op = opt.apply_gradients(g_v)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    print("init global done")
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))

```

### 3.1.4 EV Initializer

We found that the accuracy will decrease significantly if glorot uniform initializer is not used when training WDL, DIN and DIEN (e.g. AUC increases very slowly and the best AUC decreases significantly when using ones\_initializer). However, EmbeddingVariable requires a dynamic shape while glorot uniform initializer requires a static shape. Here are the methods to solve the problems caused by static shape in other frameworks:

- PAI-TF140 & 1120: Generate a default value tensor with fixed shape in each step, which will affect the performance when users enlarge the size of default value tensor for better accuracy of models.
- XDL: XDL first generates a default value matrix with static shape at initialization stage. A default value is fetched every time a new feature comes. A new default value matrix is generated when all default values are fetched. The advantage of this approach is it can promise every feature can have its unique default value. But on the other hand, this approach needs to set a mutex lock when generating the default value matrix, which will decrease performance. Moreover, the default value matrix generator in a temporarily constructed context doesn't have the info of the graph, so users can not fix the default value by setting the random seed.
- Abacus: Abacus calls the initializer individually for each feature to generate a default value, which hardly affects the performance, but there are two disadvantages of this method. First, because the initializer only generates one default value every time, default values may not conform to the distribution. Second, if users set the random seed, default values of features will be the same.

Considering the above methods, we implement EV Initializer. EV Initializer generates a default value matrix at the initialization stage. Then every new feature fetches its default value according to the index calculated by its id mod a fixed size. EV Initializer first avoids the effect of mutex lock. Second, default values will conform to distribution. Finally users can fix the default value by setting the random seed.

### Usage

Users can refer to the following examples to set EV Initializer

```

init_opt = tf.InitializerOption(initializer=tf.glorot_uniform_initializer,
                                default_value_dim = 10000)
ev_opt = tf.EmbeddingVariableOption(init_option=init_opt)

#Create EmbeddingVariable with get_embedding_variable
emb_var = tf.get_embedding_variable("var", embedding_dim = 16, ev_option=ev_opt)

#Create EmbeddingVariable with sparse_column_wth_embedding

```

(continues on next page)

(continued from previous page)

```

from tensorflow.contrib.layers.python.layers import feature_column
emb_var = feature_column.sparse_column_with_embedding("var", ev_option=ev_opt)

#Create EmbeddingVariable with categorical_column_with_embedding
emb_var = tf.feature_column.categorical_column_with_embedding("var", ev_option=ev_opt)

```

Here is the definition of `InitializerOption`

```

@tf_export(v1=["InitializerOption"])
class InitializerOption(object):
    def __init__(self,
                  initializer = None,
                  default_value_dim = 4096,
                  default_value_no_permission = .0):
        self.initializer = initializer
        self.default_value_dim = default_value_dim
        self.default_value_no_permission = default_value_no_permission
        if default_value_dim <=0:
            print("default value dim must larger than 1, the default value dim is set to_
↪default 4096.")
            default_value_dim = 4096

```

- `initializer`: the initializer of `EmbeddingVariable`, default is truncated normal initializer.
- `default_value_dim`: the number of default values generated by EV Initializer, the configuration can be referred to the hash bucket size or the number of features, the default value is 4096.
- `default_value_no_permission`: the default value for filtered features when enabling feature filter.

## 3.2 Feature Eviction of EmbeddingVariable

### 3.2.1 Introduction

For some features that are not helpful for training, we need to eliminate them so as not to affect the training effect, and also save memory. In DeepRec, we support the feature eviction, which triggers feature elimination every time checkpoint is saved. Currently, we provide two strategies for feature eviction:

- Feature eviction based on global step: The first method is to judge whether a feature should be eliminated according to the global step. We will assign a timestamp to each feature, which will be updated with the current global step each time the feature is updated in backward. When saving ckpt, it is judged whether the gap between the current global step and the timestamp exceeds a threshold, and if so, this feature is eliminated (that is, deleted). The advantage of this method is that the query and update overhead is relatively small, but the disadvantage is that an int64 data is required to record metadata, which has additional memory overhead. The user configures the threshold size of elimination by configuring the `steps_to_live` parameter.
- Feature eviction based on l2 weight: In training, if the L2 norm of the embedding value of a feature is smaller, it means that the contribution of this feature in the model is smaller. Therefore, when ckpt is saved, the L2 norm is smaller than a certain threshold. Characteristics. The advantage of this method is that no additional metadata is required, but the disadvantage is that it introduces additional computational overhead. The user configures the threshold size for elimination by configuring `l2_weight_threshold`.

### 3.2.2 API

```
@tf_export(v1=["GlobalStepEvict"])
class GlobalStepEvict(object):
    def __init__(self,
                  steps_to_live = None):
        self.steps_to_live = steps_to_live

@tf_export(v1=["L2WeightEvict"])
class L2WeightEvict(object):
    def __init__(self,
                  l2_weight_threshold = -1.0):
        self.l2_weight_threshold = l2_weight_threshold
        if l2_weight_threshold <= 0 and l2_weight_threshold != -1.0:
            print("l2_weight_threshold is invalid, l2_weight-based eviction is disabled")
```

Description:

- `steps_to_live`: The threshold for global step feature eviction, if the feature has not been visited for more than `steps_to_live` global steps, then it will be eliminated
- `l2_weight_threshold`: The threshold for L2 weight feature elimination, if the L2-norm of the feature is less than the threshold, it will be eliminated

### 3.2.3 Usage

If `GlobalStepEvict` or `L2WeightEvict` is not configured, `steps_to_live` is set to `None` and `l2_weight_threshold` is set to less than 0, then the feature is disabled, otherwise the feature is enabled.

Users can use the feature eviction through the following methods.

```
# global step feature eviction
evict_opt = tf.GlobalStepEvict(steps_to_live=4000)

# l2 weight feature eviction
evict_opt = tf.L2WeightEvict(l2_weight_threshold=1.0)

ev_opt = tf.EmbeddingVariableOption(evict_option=evict_opt)

# get_embedding_variable interface
emb_var = tf.get_embedding_variable("var", embedding_dim = 16, ev_option=ev_opt)

# sparse_column_with_embedding interface
from tensorflow.contrib.layers.python.layers import feature_column
emb_var = feature_column.sparse_column_with_embedding("var", ev_option=ev_opt)

emb_var = tf.feature_column.categorical_column_with_embedding("var", ev_option=ev_opt)
```

## 3.3 Feature Filter of EmbeddingVariable

### 3.3.1 Introduction

When the frequency of features is too low, not only will it not help the training effect of the model, but it will also cause memory waste and overfitting. In order to solve the above problems, DeepRec provides the feature filter that allows users to filter low-frequency features. Currently DeepRec supports two feature filters: counter feature filter and Bloom feature filter:

- **Counter feature filter:** The counter feature filter will record the number of times each feature is accessed in the forward process, and only the features whose frequency exceeds the threshold will be assigned to the embedding vector and updated in the backward process. The advantage of this kind of filter is that it can accurately count the number of times of each feature, and at the same time, the frequency query can be completed at the same time as querying the embedding vector, so there is almost no additional time overhead compared to when the feature filter is not used. The disadvantage is that in order to reduce the number of queries, even for features that are not allowed, it is necessary to record all the metadata of the corresponding features. When the rate of access is low, it uses more memory than the bloom feature filter.
- **Bloom feature filter:** Bloom feature filter is implemented based on Counter Bloom Filter. The advantage of this method is that it can greatly reduce memory usage when most of the features are low-frequency. The disadvantage is that multiple hashes and queries are required, which will bring obvious overhead. At the same time, when the proportion of high-frequency features is relatively high, the data structure of the Bloom feature filter will take up a lot of memory.
- **Initialization of filtered features:** When the user uses the feature filter, for the features whose frequency does not exceed the threshold, the embedding value obtained by the query at this time is the default\_value\_no\_permission set in the\_INITIALIZEROption, and the default value is 0.0.

### 3.3.2 Usage

Users can refer to the following example to use the feature filter

```
#Configure Bloom feature filter
filter_option = tf.CBFFilter(filter_freq=3,
                             max_element_size = 2**30,
                             false_positive_probability = 0.01,
                             counter_type=dtypes.int64)

#Configure Counter feature filter
filter_option = tf.CounterFilter(filter_freq=3)

ev_opt = tf.EmbeddingVariableOption(filter_option=filter_option)
#Enable feature filter with get_embedding_variable API
emb_var = get_embedding_variable("var", embedding_dim = 16, ev_option=ev_opt)

#Enable feature filter with sparse_column_with_embedding API
from tensorflow.contrib.layers.python.layers import feature_column
emb_var = feature_column.sparse_column_with_embedding("var", ev_option=ev_opt)

#Enable feature filter with categorical_column_with_embedding API
emb_var = tf.feature_column.categorical_column_with_embedding("var", ev_option=ev_opt)
```

The following is the definition of the feature filter interface:

```
@tf_export(v1=["CounterFilter"])
class CounterFilter(object):
    def __init__(self, filter_freq = 0):
        self.filter_freq = filter_freq

@tf_export(v1=["CBFFilter"])
class CBFFilter(object):
    def __init__(self,
                  filter_freq = 0,
                  max_element_size = 0,
                  false_positive_probability = -1.0,
                  counter_type = dtypes.uint64)
```

**parameters:**

- `filter_freq`: The minimum frequency that the feature needs to achieve to be trained
- `max_element_size`: The number of features estimated by the user
- `false_positive_probability`: The error rate of the bloom filter
- `counter_type`: Data type of the frequency

The parameter setting of the bloom feature filter can refer to the following table, where  $m$  is the length of the bloom filter,  $n$  is `max_element_size`,  $k$  is the number of hash functions, and the value in the table is `false_positive_probability`:



m/n	最优k	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8
2	1.39	0.393	0.400						
3	2.08	0.283	0.237	0.253					
4	2.77	0.221	0.155	0.147	0.160				
5	3.46	0.181	0.109	0.092	0.092	0.101			
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578	0.0638		
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347	0.0364		
8	5.55	0.118	0.0489	0.0306	0.024	0.0217	0.0216	0.0229	
9	6.24	0.105	0.0397	0.0228	0.0166	0.0141	0.0133	0.0135	0.0145
10	6.93	0.0952	0.0329	0.0174	0.0118	0.00943	0.00844	0.00819	0.00846
11	7.62	0.0869	0.0276	0.0136	0.00864	0.0065	0.00552	0.00513	0.00509
12	8.32	0.08	0.0236	0.0108	0.00646	0.00459	0.00371	0.00329	0.00314
13	9.01	0.074	0.0203	0.00875	0.00492	0.00332	0.00255	0.00217	0.00199
14	9.7	0.0689	0.0177	0.00718	0.00381	0.00244	0.00179	0.00146	0.00129
15	10.4	0.0645	0.0156	0.00596	0.003	0.00183	0.00128	0.001	0.000852
16	11.1	0.0606	0.0138	0.005	0.00239	0.00139	0.000935	0.000702	0.000574
17	11.8	0.0571	0.0123	0.00423	0.00193	0.00107	0.000692	0.000499	0.000394
18	12.5	0.054	0.0111	0.00362	0.00158	0.000839	0.000519	0.00036	0.000275

**Feature filter not configured:** If CounterFilter or CBFFilter is not passed in when constructing EmbeddingVariableOption object, or filter\_freq is set to 0, the feature filter is disabled.

**Checkpoint:** When using tf.train.saver, regardless of whether the frequency of the feature reaches the threshold, its id and frequency will be recorded in checkpoint, and the embedding of filtered features will not be saved in checkpoint. When loading checkpoint, for the filtered features in checkpoint, it is determined whether to be filtered in the new round of training by comparing its frequency with the threshold value of the filter. For the features that have participated in the training, no matter whether the feature frequency in checkpoint exceeds the threshold, it is considered to be a feature that has been admitted in the new round of training. At the same time, checkpoint supports forward compatibility, checkpoint without counter records can be read. Incremental checkpoint is not currently supported.

**Configure filter\_freq:** Users need to configure according to the samples.

**Feature filter and embedding multi-tier storage:** Because the bloom feature filter and the Embedding multi-tier storage are based on different counting components, opening two features at the same time will cause errors in the counting function, so it is currently invalid to use bloom feature filter and the embedding multi-tier storage at the same time.

**Collect information of filtered features:**

If a user wants to obtain information about filtered features, such as their ids and frequencies, the user can obtain this information by reading the content in the checkpoint. The method of reading the content in the checkpoint is as follows:

```
from tensorflow.contrib.framework.python.framework import checkpoint_utils
for name, shape in checkpoint_utils.list_variables("xxxxx.ckpt"):
    print('loading... ', name, shape, checkpoint_utils.load_variable("xxxxx.ckpt",
↵name))
```

```
loading... global_step [] 1
loading... var_dist-freqs [2] [3 3]
loading... var_dist-freqs_filtered [2] [2 1]
loading... var_dist-keys [2] [1 2]
loading... var_dist-keys_filtered [2] [3 4]
loading... var_dist-partition_filter_offset [1001] [0 0 0
loading... var_dist-partition_offset [1001] [0 0 1 ... 2 2
loading... var_dist-values [2, 4] [[0.00078034 0.00078034 0
[0.00138596 0.00138596 0.00138596 0.00138596]]
loading... var_dist-versions [0] []
loading... var_dist-versions_filtered [0] []
loading... var_dist/Adagrad-freqs [2] [3 3]
loading... var_dist/Adagrad-freqs_filtered [2] [2 1]
loading... var_dist/Adagrad-keys [2] [1 2]
loading... var_dist/Adagrad-keys_filtered [2] [3 4]
loading... var_dist/Adagrad-partition_filter_offset [1001]
loading... var_dist/Adagrad-partition_offset [1001] [0 0 1
loading... var_dist/Adagrad-values [2, 4] [[64.1 64.1 64.1
[36.1 36.1 36.1 36.1]]
loading... var_dist/Adagrad-versions [0] []
loading... var_dist/Adagrad-versions_filtered [0] []
```

Execute the above code to get the following results:

For an EmbeddingVariable, it will be divided into 9 parts when it is stored in checkpoint. Assuming that the name of the EmbeddingVariable is var, users will see in checkpoint:

- var-keys: Ids of the unfiltered features
- var-values: Embeddings of the unfiltered features
- var-freqs: Frequencies of the unfiltered features
- var-versions: Last updated step of the unfiltered features
- var-keys\_filtered: Ids of the filtered features
- var-freqs\_filtered: Frequencies of the filtered features
- var-versions\_filtered: Last updated step of the filtered features
- var-partition\_offset: Parameters for restoring unfiltered features
- var-partition\_filter\_offset: Parameters for restoring filtered features

Users can collect information about filtered features by reading data with `_filtered` suffix.

### Whether to save information of filtered features

Users sometimes don't need to save the information of filtered features when saving the ckpt (for example, the information of filtered features is not needed during serving). Users can set the environment variable `TF_EV_SAVE_FILTERED_FEATURES` to `False` to not save the information of the filtered features, thus reducing the size of the checkpoint.

### Remove the information of filtered features in checkpoint

For the model that is about to serve online, the information of the filtered features in the checkpoint is useless, therefore DeepRec provides tools that can remove the information of filtered features in the checkpoint. After compiling and

installing DeepRec, users can use the tool with the following command:

```
python Deeprec/tensorflow/python/tools/shrink_ckpt_with_filtered_features.py --input_
↪checkpoint /root/code/model.ckpt --output_checkpoint /root/shrink.ckpt
```

The tool has the following parameters:

- `input_checkpoint`: The full path of the input checkpoint, required
- `output_checkpoint`: The full path of the output checkpoint, required

## 3.4 Adaptive Embedding

### 3.4.1 Introduction

If there are many sparse features, it will cause `EmbeddingVariable` (abbreviate as EV) to occupy lots of memory. On the one hand, it will put a lot of pressure on PS memory during training. On the other hand, when the model is served online, the model needs to be shrunk. The usual method is to filter features through a static filter. However, this filtering rule is too rigid, and the embedding in dynamic learning is not well considered, which will inevitably affect the final accuracy. Moreover, due to the different frequencies of features, and EV starts to learn all features from the initial value set by the initializer (generally set to 0), it will learn very slowly for features that do not appear very much. For some features with low to high frequency of occurrence, it is also necessary to gradually learn to a better state, and the learning results of other features cannot be shared. This is caused by the conflict-free thinking of EV.

Adaptive Embedding uses a static Variable and a dynamic EV to store sparse features together. For low-frequency features, it is stored in conflictable static Variable, and for features with high frequency, it is stored in non-conflicting EV. Migrating embedding to EV can reuse the learning results in the static Variable, which greatly reduces the model size.

### 3.4.2 API

Adaptive Embedding needs to be used through the `feature_column` API

```
def categorical_column_with_adaptive_embedding(key,
                                             hash_bucket_size,
                                             dtype=dtypes.string,
                                             partition_num=None,
                                             ev_option=variables.
↪EmbeddingVariableOption())

# key, name of feature_column
# hash_bucket_size, the first dimension of static Variable
# dtype, data type of sparse features
# partition_num, partition number of EV, no partition by default
# ev_option, configuration of EV
```

### 3.4.3 Example

```
import tensorflow as tf

columns = tf.feature_column.categorical_column_with_adaptive_embedding("col_emb", hash_
↳ bucket_size=100, dtype=tf.int64)
W = tf.feature_column.embedding_column(categorical_column=columns,
                                     dimension=3,
                                     initializer=tf.ones_initializer(tf.float32))

ids={}
ids["col_emb"] = tf.SparseTensor(indices=[[0,0],[1,0],[2,0],[3,0],[4,0],[5,0],[6,0],[7,
↳ 0],[8,0],[9,0]],
                                values=tf.cast([1,2,3,4,5,6,7,8,9,0], tf.int64),
                                dense_shape=[10, 1])

adaptive_mask_tensors={}
adaptive_mask_tensors["col_emb"] = tf.cast([1,0,1,0,1,0,0,1,0,1], tf.int32)
emb = tf.feature_column.input_layer(ids, [W], adaptive_mask_tensors=adaptive_mask_
↳ tensors)

fun = tf.multiply(emb, 2.0, name='multiply')
loss = tf.reduce_sum(fun, name='reduce_sum')

graph=tf.get_default_graph()

opt = tf.train.FtrlOptimizer(0.1, l1_regularization_strength=2.0, l2_regularization_
↳ strength=0.00001)
g_v = opt.compute_gradients(loss)
train_op = opt.apply_gradients(g_v)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run([init])
    emb1, top, l = sess.run([emb, train_op, loss])
    emb1, top, l = sess.run([emb, train_op, loss])
    emb1, top, l = sess.run([emb, train_op, loss])
    emb1, top, l = sess.run([emb, train_op, loss])
    print(emb1, top, l)
```

## 3.5 Multi-Hash Variable

### 3.5.1 Background

In the deep learning scenario, in order to train the features of the ID class (such as user id or item id), the sparse features of the id class are usually mapped to the corresponding low-dimensional dense embedding vectors. In the recommendation scenario, as the amount of data gradually increases, a corresponding embedding vector is stored for each id class feature, that is, the non-conflicting Hash scheme (EmbeddingVariable) will cause the model to be larger than the model under the conflicting Hash method. (Variable). In order to solve the Training/Inference performance and resource overhead caused by larger models, DeepRec has provided several ways to solve performance and resource problems. 1) Feature elimination, by eliminating and cleaning outdated features, avoiding excessive accumulation of expired features, wastes memory; 2) feature access, low-frequency feature access mechanism, avoiding low-frequency feature overfitting while reducing memory usage; 3) mixed multi-level EmbeddingVariable, supported by storing un-

popular features in cheap and larger storage media Larger models; 4) DynamicDimensionEmbeddingVariable, by using strategies to allow unpopular features to be expressed with smaller dims to reduce the storage space occupied by unpopular features. The Multi-Hash Variable introduced in this article is another way to reduce memory/video memory usage.

《Compositional Embeddings Using Complementary Partitions for Memory-Efficient Recommendation Systems》 proposes a new way to reduce memory usage, the specific method is:

1. Construct several small Embedding tables
2. Each corresponding table corresponds to a hash function, and these hash functions need to be complementary, that is, for each id, its corresponding hash value set is unique, and is not exactly the same as any other id. For example, when there are two embedding tables, using Quotient-Reminder can ensure that each key has a unique hash set, as shown in the following figure:

## (1) Naive Complementary Partition: If

$$P = \{\{x\} : x \in S\}$$

then  $P$  is a complementary partition by definition. This corresponds to a full embedding table with dimension  $|S| \times D$ .

(2) Quotient-Remainder Complementary Partitions: Given  $m \in \mathbb{N}$ , the partitions

$$P_1 = \{\{x \in S : \varepsilon(x) \setminus m = l\} : l \in \mathcal{E}(\lceil |S|/m \rceil)\}$$

$$P_2 = \{\{x \in S : \varepsilon(x) \bmod m = l\} : l \in \mathcal{E}(m)\}$$

are complementary. This corresponds to the quotient-remainder trick in Section 2.

(3) Generalized Quotient-Remainder Complementary Partitions: Given  $m_i \in \mathbb{N}$  for  $i = 1, \dots, k$  such that  $|S| \leq \prod_{i=1}^k m_i$ , we can recursively define complementary partitions

$$P_1 = \{\{x \in S : \varepsilon(x) \bmod m_1 = l\} : l \in \mathcal{E}(m_1)\}$$

$$P_j = \{\{x \in S : \varepsilon(x) \setminus M_j \bmod m_j = l\} : l \in \mathcal{E}(m_j)\}$$

where  $M_j = \prod_{i=1}^{j-1} m_i$  for  $j = 2, \dots, k$ . This generalizes the quotient-remainder trick.

(4) Chinese Remainder Partitions: Consider a pairwise coprime factorization greater than or equal to  $|S|$ , that is,  $|S| \leq \prod_{i=1}^k m_i$  for  $m_i \in \mathbb{N}$  for all  $i = 1, \dots, k$  and  $\gcd(m_i, m_j) = 1$  for all  $i \neq j$ . Then we can define the complementary partitions

$$P_j = \{\{x \in S : \varepsilon(x) \bmod m_j = l\} : l \in \mathcal{E}(m_j)\}$$

for  $j = 1, \dots, k$ .

3. According to a certain strategy, the embeddings taken from multiple tables are combined into the final embedding, such as add, multiply, and concat.

### 3.5.2 Multi-Hash Variable

In DeepRec, we implemented the Multi-Hash Variable. You can use this feature through `get_multihash_variable` interface as follows:

```
def get_multihash_variable(name,
                           dims,
                           num_of_partitions=2,
                           complementary_strategy="Q-R",
                           operation="add",
                           dtype=float,
                           initializer=None,
                           regularizer=None,
                           trainable=None,
                           collections=None,
                           caching_device=None,
                           partitioner=None,
                           validate_shape=True,
                           use_resource = None,
                           custom_getter=None,
                           constraint=None,
                           synchronization=VariableSynchronization.AUTO,
                           aggregation=VariableAggregation.NONE):

# name: multihash variable's name
# embedding dim: A list needs to be passed in. If the length of the list is 1, the
    ↳ operation must be selected from add or mult;
                If the length of the list is greater than 1, then the operation must
    ↳ choose concat, and the total length of the elements in the list must be equal to
    ↳ embedding_dim
# num_of_partitions: The number of variable partitions. If complementary_strategy is "Q-R",
    ↳ then the
                Argument must be 2.
# complementary_strategy: "Q-R" is supported
# operation: one of "add", "mult" and "concat"
# initializer: same as variable
# partitioner: same as variable
```

Only two partitions are supported for multi-hashing using the QR method. The reason is that according to the experiments in the paper, this method can already support most scenarios. At the same time, the method of more than three partitions is relatively complicated and will bring more problems. Much lookup overhead.



### 3.5.3 Example

Use interface `get_multihash_variable`

```
import tensorflow as tf

def main(unused_argv):
    embedding = tf.get_multihash_variable("var-dist",
                                         [[2,2],[2,2]],
                                         complementary_strategy="Q-R",
                                         operation="concat",
                                         initializer=tf.ones_initializer)

    var = tf.nn.embedding_lookup(embedding, [0,1,2,3])
    fun = tf.multiply(var, 2.0, name='multiply')
    loss1 = tf.reduce_sum(fun, name='reduce_sum')
    opt = tf.train.AdagradOptimizer(0.1)
    g_v = opt.compute_gradients(loss1)
    train_op = opt.apply_gradients(g_v)
    init = tf.global_variables_initializer()

    with tf.Session() as sess:
        sess.run([init])
        print(sess.run([var, train_op]))
        print(sess.run([var, train_op]))
        print(sess.run([var, train_op]))

if __name__=="__main__":
    tf.app.run()
```

Use interface `categorical_column_with_multihash`

```
import tensorflow as tf
from tensorflow.python.framework import ops
from tensorflow.python.feature_column import feature_column_v2 as fc2

columns = fc2.categorical_column_with_multihash("col_emb", dims = (2,2))
W = tf.feature_column.embedding_column(categorical_column=columns,
                                     dimension=(2,3),
                                     initializer=tf.ones_initializer(tf.dtypes.float32))

ids={}
ids["col_emb"] = tf.SparseTensor(indices=[[0,0],[1,1],[2,2],[3,3]], values=tf.cast([0,1,
↪ 2,3], tf.dtypes.int64), dense_shape=[4, 4])

emb = tf.feature_column.input_layer(ids, [W])
fun = tf.multiply(emb, 2.0, name='multiply')
loss1 = tf.reduce_sum(fun, name='reduce_sum')
opt = tf.train.AdagradOptimizer(0.1)
g_v = opt.compute_gradients(loss1)
train_op = opt.apply_gradients(g_v)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
```

(continues on next page)



(continued from previous page)

```
print("init global done")
print(sess.run([emb, train_op]))
print(sess.run([emb, train_op]))
print(sess.run([emb, train_op]))
```

## 3.6 Group Embedding

### 3.6.1 Background

DeepRec has done lots of Op Fusion in single EmbeddingVariable Subgraph(Mostly fuse used-less ops introduced by `tf.nn.embedding_lookup_sparse`) which outperform the native Tensorflow Runtime. However, in a typical CTR scenario, there will actually be hundreds of EmbeddingLookups. Through profiling, we found that the problem of multiple Op kernels launch or CUDA kernels brought by EmbeddingLookups is the bottleneck. So, we hope to develop a function of simultaneously aggregation of multiple EmbeddingLookup, so as to improve EmbeddingLookup performance of this scene.

The Group Embedding functions supports simultaneously aggregated multiple EmbeddingLookups on GPUs or CPUs. This function is designed to support single-card Fusion and multi-card Fusion which is shown below:

API	Device Type	Note
localized	CPU/GPU	Single-card training or serving is recommended.
collective	GPU	Multi-card training is recommended.
parameter_server(WIP)	-	Currently not supported.

### 3.6.2 Localized training mode

#### User API

GroupEmbedding provides two levels of API. The one is `tf.nn.group_embedding_lookup_sparse` & `tf.nn.group_embedding_lookup` and the other is `tf.feature_column.group_embedding_column_scope` which is based on `feature_column` API.

#### `group_embedding_lookup_sparse`

```
def group_embedding_lookup_sparse(params,
                                sp_ids,
                                combiners,
                                sp_weights=None,
                                partition_strategy="mod",
                                is_sequence=False,
                                params_num_per_group=sys.maxsize,
                                name=None):
```

- `params` : List, This parameter could receive one or more EmbeddingVariables or native Tensorflow Variable.
- `sp_ids` : List | Tuple, SparseTensor `sp_ids` is the ID used for EmbeddingLookup, the length must be consistent with `params`.
- `combiners` : List | Tuple, The pooling method of embedding values. Currently support mean and sum.
- `sp_weights` : List | Tuple the weight of `sp_ids` values.

- `partition_strategy` : str, Currently not supported.
- `is_sequence` : bool, Op would return Tensor shape of [B, T, D] if True
- `params_num_per_group` : int, This parameter indicates the number of Variables inside each Op. The default setting is the maximum value. The default value is suitable for GPU scenarios; when using the CPU, it is recommended to set the smaller the better.
- `name` : str group name

#### `group_embedding_lookup`

```
def group_embedding_lookup(params,
                           ids,
                           partition_strategy="mod",
                           name=None):
```

- `params` : List, This parameter could receive one or more EmbeddingVariables or native Tensorflow Variable.
- `ids` : List | Tuple, Tensor ids is the ID used for EmbeddingLookup, the length must be consistent with `params`.
- `partition_strategy` : str, Currently not supported.
- `name` : str group name

#### `group_embedding_column_scope`

```
def group_embedding_column_scope(name=None):
```

- `name` : The name of scope.

We need to initialize a context `group_embedding_column_scope`, and complete the construction of `EmbeddingColumn` in that context. Later, the `EmbeddingColumn Lookup` would be simultaneously aggregate by `tf.feature_column.input_layer`. It is worth noting that the underlying implementation of this function is designed for `tf.SparseTensor`.

### Example

#### Usage of `group_embedding_lookup_sparse`

```
import tensorflow as tf

ev_opt = tf.EmbeddingVariableOption(evict_option=None,
                                    filter_option=None)

with tf.device('/GPU:{}'.format(0)): # place EV on CPU if using CPU GroupEmbedding
    var_0 = tf.get_embedding_variable("var_0",
                                     embedding_dim=16,
                                     initializer=tf.ones_initializer(tf.float32),
                                     ev_option=ev_opt)

    var_1 = tf.get_embedding_variable("var_1",
                                     embedding_dim=8,
                                     initializer=tf.ones_initializer(tf.float32),
                                     ev_option=ev_opt)

##We recommend use RaggedTensor representation here.
indices_0 = tf.RaggedTensor.from_row_splits(
```

(continues on next page)

(continued from previous page)

```

values=[3, 1, 4, 1, 5, 9, 2, 6],
row_splits=[0, 4, 4, 7, 8, 8])

embedding_weights = [var_0, var_1]
indices = [indices_0 for _ in range(2)]
combiners = ["sum", "sum"]

deep_features = tf.nn.group_embedding_lookup_sparse(embedding_weights, indices,
↳combiners)

init = tf.global_variables_initializer()
sess_config = tf.ConfigProto()
sess_config.gpu_options.visible_device_list = "0" #str(hvd.local_rank())
sess_config.gpu_options.allow_growth = True
with tf.Session(config=sess_config) as sess:
    sess.run(init)
    print("init global done")
    print(sess.run([deep_features]))

```

### Usage of feature\_column

```

from tensorflow.python.feature_column import feature_column as fc_old
from tensorflow.python.framework import dtypes

ev_opt = tf.EmbeddingVariableOption(evict_option=None,
                                   filter_option=None)

# group_name represent for the grouped embedding variable
with tf.feature_column.group_embedding_column_scope(name="item")::
    ad0_col = tf.feature_column.categorical_column_with_embedding(
        key='ad0', dtype=dtypes.int64, ev_option=ev_opt)
    ad0_fc = tf.feature_column.embedding_column(
        categorical_column=ad0_col,
        dimension=20,
        initializer=tf.constant_initializer(0.5))
    ad1_fc = tf.feature_column.embedding_column(
        tf.feature_column.categorical_column_with_embedding(
            key='ad1', dtype=dtypes.int64, ev_option=ev_opt),
        dimension=30,
        initializer=tf.constant_initializer(0.5))

with tf.feature_column.group_embedding_column_scope(name="user")::
    user0_fc = tf.feature_column.embedding_column(
        tf.feature_column.categorical_column_with_embedding(
            key='user0', dtype=dtypes.int64, ev_option=ev_opt),
        dimension=20,
        initializer=tf.constant_initializer(0.5))

columns = [ad0_fc, ad1_fc, user0_fc]

ids={}
ids["ad0"] = tf.SparseTensor(indices=[[0,0],[0,1],[1,1],[2,2],[3,3],[4,3],[4,4]], \

```

(continues on next page)

(continued from previous page)

```

        values=tf.cast([1,3,2,3,4,5,3], tf.dtypes.int64), dense_shape=[5,
↪ 5])
ids["ad1"] = tf.SparseTensor(indices=[[0,0],[0,1],[1,1],[2,2],[3,3],[4,3],[4,4]], \
        values=tf.cast([1,3,2,3,4,5,3], tf.dtypes.int64), dense_shape=[5,
↪ 5])
ids["user0"] = tf.SparseTensor(indices=[[0,0],[0,1],[1,1],[2,2],[3,3],[4,3],[4,4]], \
        values=tf.cast([1,3,2,3,4,5,3], tf.dtypes.int64), dense_shape=[5,
↪ 5])
with tf.device('/gpu:0'):
    emb = tf.feature_column.input_layer(ids, columns)
    fun = tf.multiply(emb, 2.0, name='multiply')
    loss = tf.reduce_sum(fun, name='reduce_sum')
    opt = tf.train.FtrlOptimizer(0.1, l1_regularization_strength=2.0, l2_regularization_
↪ strength=0.00001)
    g_v = opt.compute_gradients(loss)
    train_op = opt.apply_gradients(g_v)
    init = tf.global_variables_initializer()

sess_config = tf.ConfigProto()
sess_config.gpu_options.visible_device_list = "0" #(hvd.local_rank())
sess_config.gpu_options.allow_growth = True
with tf.Session(config=sess_config) as sess:
    sess.run(init)
    print("init global done")
    print(sess.run([fun, train_op,loss]))

```

### 3.6.3 Distributed training mode

#### Environment configuration

First of all, we need to ensure that SOK module is compiled.

```

bazel --output_base /tmp build -j 16 -c opt --config=opt //tensorflow/tools/pip_
↪ package:build_sok && ./bazel-bin/tensorflow/tools/pip_package/build_sok

```

#### User API

- Before using Group Embedding API, you need to enable the setting `tf.config.experimental.enable_distributed_strategy()` parameters setting:
  - `strategy="collective"`. This mode will complete the initialization of the horovod module and SOK-related dependencies and is recommended to be used in single worker multi card Training.
  - `strategy="parameter_server"`. This mode is recommended to be used in ps worker Distributed Training.(WIP)
- Distributed training mode also provide two levels of API. The one is `tf.nn.group_embedding_lookup_sparse` and the other is `tf.feature_column.group_embedding_column_scope` which is based on `feature_column` API. **Note:** The only difference between the above two modes in the use interface is that the `sp_id` of distributed training mode is suitable for the input of `RaggedTensor` while localized training mode is suitable for the input of `SparseTensor`

## Example

### Usage of group\_embedding\_lookup\_sparse

```
import tensorflow as tf

tf.config.experimental.enable_distributed_strategy(strategy="collective")

ev_opt = tf.EmbeddingVariableOption(evict_option=None,
                                    filter_option=None)

with tf.device('/GPU:{}'.format(0)):
    var_0 = tf.get_embedding_variable("var_0",
                                     embedding_dim=16,
                                     initializer=tf.ones_initializer(tf.float32),
                                     ev_option=ev_opt)

    var_1 = tf.get_embedding_variable("var_1",
                                     embedding_dim=8,
                                     initializer=tf.ones_initializer(tf.float32),
                                     ev_option=ev_opt)

##We recommend use RaggedTensor representation here.
indices_0 = tf.RaggedTensor.from_row_splits(
    values=[3, 1, 4, 1, 5, 9, 2, 6],
    row_splits=[0, 4, 4, 7, 8, 8])

embedding_weights = [var_0, var_1]
indices = [indices_0 for _ in range(2)]
combiners = ["sum", "sum"]

deep_features = tf.nn.group_embedding_lookup_sparse(embedding_weights, indices,
                                                    combiners)

init = tf.global_variables_initializer()
sess_config = tf.ConfigProto()
sess_config.gpu_options.visible_device_list = "0" #str(hvd.local_rank())
sess_config.gpu_options.allow_growth = True
with tf.Session(config=sess_config) as sess:
    sess.run(init)
    print("init global done")
    print(sess.run([deep_features]))
```

### Usage of feature\_column

```
from tensorflow.python.feature_column import feature_column as fc_old
from tensorflow.python.framework import dtypes
tf.config.experimental.enable_distributed_strategy(strategy="collective")

ev_opt = tf.EmbeddingVariableOption(evict_option=None,
                                    filter_option=None)

# group_name represent for the grouped embedding variable
with tf.feature_column.group_embedding_column_scope(name="item")::
```

(continues on next page)

(continued from previous page)

```

ad0_col = tf.feature_column.categorical_column_with_embedding(
    key='ad0', dtype=dtypes.int64, ev_option=ev_opt)
ad0_fc = tf.feature_column.embedding_column(
    categorical_column=ad0_col,
    dimension=20,
    initializer=tf.constant_initializer(0.5))
ad1_fc = tf.feature_column.embedding_column(
    tf.feature_column.categorical_column_with_embedding(
        key='ad1', dtype=dtypes.int64, ev_option=ev_opt),
    dimension=30,
    initializer=tf.constant_initializer(0.5))

with tf.feature_column.group_embedding_column_scope(name="user")::
    user0_fc = tf.feature_column.embedding_column(
        tf.feature_column.categorical_column_with_embedding(
            key='user0', dtype=dtypes.int64, ev_option=ev_opt),
        dimension=20,
        initializer=tf.constant_initializer(0.5))

columns = [ad0_fc, ad1_fc, user0_fc]

ids={}
ids["ad0"] = tf.SparseTensor(indices=[[0,0],[0,1],[1,1],[2,2],[3,3],[4,3],[4,4]], \
    values=tf.cast([1,3,2,3,4,5,3], tf.dtypes.int64), dense_shape=[5,
    ↪ 5])
ids["ad1"] = tf.SparseTensor(indices=[[0,0],[0,1],[1,1],[2,2],[3,3],[4,3],[4,4]], \
    values=tf.cast([1,3,2,3,4,5,3], tf.dtypes.int64), dense_shape=[5,
    ↪ 5])
ids["user0"] = tf.SparseTensor(indices=[[0,0],[0,1],[1,1],[2,2],[3,3],[4,3],[4,4]], \
    values=tf.cast([1,3,2,3,4,5,3], tf.dtypes.int64), dense_shape=[5,
    ↪ 5])

with tf.device('/GPU:{}'.format(0)):
    emb = tf.feature_column.input_layer(ids, columns)
    fun = tf.multiply(emb, 2.0, name='multiply')
    loss = tf.reduce_sum(fun, name='reduce_sum')
    opt = tf.train.FtrlOptimizer(0.1, l1_regularization_strength=2.0, l2_regularization_
    ↪ strength=0.00001)
    g_v = opt.compute_gradients(loss)
    train_op = opt.apply_gradients(g_v)
    init = tf.global_variables_initializer()

sess_config = tf.ConfigProto()
sess_config.gpu_options.visible_device_list = "0" #(hvd.local_rank())
sess_config.gpu_options.allow_growth = True
with tf.Session(config=sess_config) as sess:
    sess.run(init)
    print("init global done")
    print(sess.run([fun, train_op, loss]))

```

More detailed usage of group embedding lookup please refer to modelzooDCNv2模型示例

## Benchmarks

We evaluate performance with DCNv2 model. The training dataset is Criteo and the batch\_size is 512.

collective

API	Global-step/s	Note
tf.nn.group_embedding_lookup_sparse	85.3 (+/- 0.1)	1 card
tf.nn.group_embedding_lookup_sparse	122.2 (+/- 0.2)	2 card
tf.nn.group_embedding_lookup_sparse	212.4 (+/- 0.4)	4 card
tf.feature_column.group_embedding_column_scope	79.7 (+/- 0.1)	1 card
tf.feature_column.group_embedding_column_scope	152.2 (+/- 0.2)	2 card
tf.feature_column.group_embedding_column_scope	272 (+/- 0.4)	4 card

localized | tf.nn.group\_embedding\_lookup\_sparse(Variable) | 153.2 (+/- 0.1) | 1 card |

## 3.7 GRPC++

### 3.7.1 Introduction

In a large-scale training scenario, users use a large number of workers and ps, resulting in high overhead communication. Tensorflow uses GRPC as a communication protocol, and it is difficult to meet large-scale scenarios (over hundreds or thousands of workers).

To solve above issues, DeepRec provides GRPC++ to support larger-scale training tasks. Based on Sharing-Nothing architecture, BusyPolling, zero copy, Send/Recv Fusion and so on, GRPC++ could greatly improve communication performance and provide much better performance compared with GRPC. GRPC++ supports larger training scale and better training performance, and improves the performance several times in some typical business scenarios compared with GRPC.

### 3.7.2 User API

Enabling the GRPC++ is as simple as using GRPC, only need to configure the `Protocol` field. In some scenarios, especially when there are a lot of Send/Recv operators, configure the `tensor_fuse` field in `config` to enable Send/Recv Ops fusion to avoid too many small packets. **tensor\_fuse is disabled by default.**

*tensor\_fuse could be used with GRPC as well, which also could brings performance improvement.*

### Configure GRPC++

We use seastar as communication framework in GRPC++, and also keep GRPC used by MasterSession. So when enable GRPC++, need to configure another set of ports for seastar. Configure `.endpoint_map` (filename) in execution directory, content as follows:

```
127.0.0.1:3333=127.0.0.1:5555
127.0.0.1:4444=127.0.0.1:6666
```

3333 and 4444 are GRPC ports, and 5555 and 6666 are corresponded seastar ports.

For example, 127.0.0.1:3333 is worker 0 GRPC port, and 127.0.0.1:5555 is worker 0 seastar ports.

## MonitoredTrainingSession

```
cluster = tf.train.ClusterSpec({"ps": ps_hosts, "worker": worker_hosts})

server = tf.train.Server(cluster,
                        job_name=FLAGS.job_name,
                        task_index=FLAGS.task_index,
                        protocol="grpc++")
...

with tf.train.MonitoredTrainingSession(
    master=target,
    config=tf.ConfigProto(tensor_fuse=True, # Enable Send/Recv Ops Fusion
                        ...),
    ) as mon_sess:
    ...
```

## Estimator

Use GRPC++ in Estimator, Need to setup RunConfig with protocol="grpc++":

```
session_config = tf.ConfigProto(
    tensor_fuse=True, # Enable Send/Recv Ops Fusion
    inter_op_parallelism_threads=16,
    intra_op_parallelism_threads=16)

run_config = tf.estimator.RunConfig(model_dir=model_dir,
                                    save_summary_steps=train_save_summary_steps,
                                    protocol="grpc++",
                                    session_config=session_config)

...

classifier = tf.estimator.Estimator(
    model_fn=model_fn,
    params={...},
    config=run_config) # run_config
```

*Should not use ParameterServerStrategy, which is not supported by GRPC++.*

### 3.7.3 Best Practise

In GRPC++, We provide list of environments to tune performance.

```
os.environ['WORKER_ENABLE_POLLING'] = "False"
os.environ['PS_ENABLE_POLLING'] = "False"
```

WORKER\_ENABLE\_POLLING/PS\_ENABLE\_POLLING true/false means that enable or disable communication threads polling.

```
os.environ['NETWORK_PS_CORE_NUMBER'] = "8"
os.environ['NETWORK_WORKER_CORE_NUMBER'] = "2"
```



NETWORK\_PS\_CORE\_NUMBER is used to setup Parameter Server communication thread number. NETWORK\_WORKER\_CORE\_NUMBER is used to setup Worker communication thread number.

More ParameterServer or Worker need to setup more communication thread number and enable polling.

Currently default communication thread number is Min(16, connections), default thread number is not the best.

- NETWORK\_WORKER\_CORE\_NUMBER is 2-4, because we suggest should not setup too much Parameter Server.
- NETWORK\_WORKER\_CORE\_NUMBER is 8-16, for hundreds of workers, 8-10 thread number is enough(if there is 24 core available). Need to reserve enough cores for compute thread.

```
os.environ["WORKER_DISABLE_PIN_CORES"] = "True"
os.environ["PS_DISABLE_PIN_CORES"] = "True"
```

WORKER\_DISABLE\_PIN\_CORES: communication threads pin cpu core or not in Worker, not pin core by default.

PS\_DISABLE\_PIN\_CORES: communication threads pin cpu core or not in Parameter Server, not pin core by default.

## 3.8 StarServer

### 3.8.1 Introduction

In large-scale job (hundreds or even thousands of workers), some problems in the TensorFlow are exposed, such as inefficient thread pool scheduling, lock overhead on multiple critical paths, inefficient execution engine, The overhead caused by frequent rpc and low memory usage efficiency, etc.

In order to solve the problems encountered by users in large-scale scenarios, we provide the StarServer. In StarServer we optimize graph, threadpool, executor, and memory optimization. Change the send/recv semantics in TensorFlow to pull/push semantics, and support this semantics in subgraph division. At the same time, the lock free in the process of graph execution is supported, which greatly improves the concurrent execution efficiency. StarServer perform better than grpc/grpc++ in larger scale job (with thousands of workers). In StarServer we optimizes the runtime of ParameterServer with share-nothing architecture and lock-free graph execution.

### 3.8.2 User API

Enabling the StarServer is as simple as using GRPC, only need to configure the Protocol field.

In DeepRec there are two StarServer implementation, the protocol are "star\_server" and "star\_server\_lite". Difference between the two protocol is "star\_server\_lite" use more aggressive graph partition strategy which would bring better performance but probabaly fails in some graph with RNN structure. We suggest "star\_server" which could support all scenarios.

#### Configure StarServer

We use seastar as communication framework in StarServer, and also keep GRPC used by MasterSession. So when enable StarServer, need to configure another set of ports for seastar. Configure .endpoint\_map (filename) in execution directory, content as follows:

```
127.0.0.1:3333=127.0.0.1:5555
127.0.0.1:4444=127.0.0.1:6666
```

3333 and 4444 are GRPC ports, and 5555 and 6666 are corresponded seastar ports.

For example, 127.0.0.1:3333 is worker 0 GRPC port, and 127.0.0.1:5555 is worker 0 seastar ports.

### MonitoredTrainingSession

```
cluster = tf.train.ClusterSpec({"ps": ps_hosts, "worker": worker_hosts})

server = tf.train.Server(cluster,
                        job_name=FLAGS.job_name,
                        task_index=FLAGS.task_index,
                        protocol="star_server")

...

with tf.train.MonitoredTrainingSession(
    master=target,
) as mon_sess:
    ...
```

### Estimator

Use StarServer in Estimator, Need to setup RunConfig with protocol="star\_server":

```
session_config = tf.ConfigProto(
    inter_op_parallelism_threads=16,
    intra_op_parallelism_threads=16)

run_config = tf.estimator.RunConfig(model_dir=model_dir,
                                    save_summary_steps=train_save_summary_steps,
                                    protocol="star_server",
                                    session_config=session_config)

...

classifier = tf.estimator.Estimator(
    model_fn=model_fn,
    params={...},
    config=run_config) # 配置 run_config
```

*Should not use ParameterServerStrategy, which is not supported by GRPC++.*

### 3.8.3 Best Practise

We suggest worker/ps number should be 8:1-10:1 which means 100 workers with 10 ParameterServer when use StarServer. Because StarServer provide powerful ParameterServer, which could support more workers. Besides, we provide list of enviroments to tune performance.

```
os.environ['WORKER_ENABLE_POLLING'] = "False"
os.environ['PS_ENABLE_POLLING'] = "False"
```

WORKER\_ENABLE\_POLLING/PS\_ENABLE\_POLLING true/false means that enable or disable communication threads polling.

```
os.environ['NETWORK_PS_CORE_NUMBER'] = "8"
os.environ['NETWORK_WORKER_CORE_NUMBER'] = "2"
```

NETWORK\_PS\_CORE\_NUMBER is used to setup Parameter Server communication thread number. NETWORK\_WORKER\_CORE\_NUMBER is used to setup Worker communication thread number.

More ParameterServer or Worker need to setup more communication thread number and enable polling.

Currently default communication thread number is Min(16, connections), default thread number is not the best.

- NETWORK\_WORKER\_CORE\_NUMBER is 2-4, because we suggest should not setup too much Parameter Server.
- NETWORK\_WORKER\_CORE\_NUMBER is 8-16, for hundreds of workers, 8-10 thread number is enough(if there is 24 core available). Need to reserve enough cores for compute thread.

```
os.environ["WORKER_DISABLE_PIN_CORES"] = "True"
os.environ["PS_DISABLE_PIN_CORES"] = "True"
```

WORKER\_DISABLE\_PIN\_CORES: communication threads pin cpu core or not in Worker, not pin core by default.

PS\_DISABLE\_PIN\_CORES: communication threads pin cpu core or not in Parameter Server, not pin core by default.

## 3.9 Distributed Synchronous Training-SOK

### 3.9.1 Introduction

Sparse Operation Kit (SOK) is a Python package wrapped GPU accelerated operations dedicated for sparse training / inference cases. It is designed to be compatible with common deep learning (DL) frameworks like TensorFlow.

### 3.9.2 Features

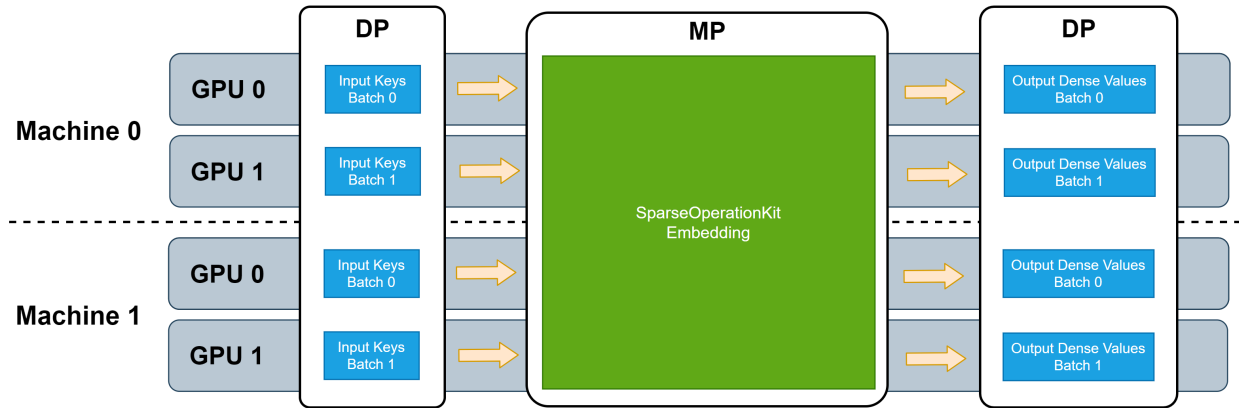
#### Model-Parallelism GPU Embedding Layer

In sparse training / inference scenarios, for instance, CTR estimation, there are vast amounts of parameters which cannot fit into the memory of a single GPU. Many common DL frameworks only offer limited support for model parallelism (MP), because it can complicate using all available GPUs in a cluster to accelerate the whole training process.

SOK provides broad MP functionality to fully utilize all available GPUs, regardless of whether these GPUs are located in a single machine or multiple machines. Simultaneously, SOK takes advantage of existing data-parallel (DP) capabilities of DL frameworks to accelerate training while minimizing code changes. With SOK embedding layers, you can build a DNN model with mixed MP and DP. MP is used to shard large embedding parameter tables, such that they are distributed among the available GPUs to balance the workload, while DP is used for layers that only consume little GPU resources.

SOK provides multiple types of MP embedding layers, optimized for different application scenarios. These embedding layers can leverage all available GPU memory in your cluster to store/retrieve embedding parameters. As a result, all utilized GPUs work synchronously.

SOK is compatible with DP training provided by common synchronized training frameworks, such as Horovod. Because the input data fed to these embedding layers can take advantage of DP, additional DP from/to MP transformations are needed when SOK is used to scale up your DNN model from single GPU to multiple GPUs. The following picture illustrates the workflow of these embedding layers.



### 3.9.3 API

- SOK is compatible with Horovod as a communication tool. First, it is initialized through the initialization function of SOK, and then two different embedding layers of sok are selected for embedding. In the process of backpropagation, we could select the optimizer optimized by SOK and some additional functions in Utilizers.
- Users can use either GroupEmbedding API in DeepRec, or use the original interface of SOK.

### GroupEmbedding API in DeepRec

#### Initialize

The GroupEmbedding interface provides localized and distributed training modes. We can use SOK in the distributed mode, first we need to enable the settings `tf.config.experimental.enable_distributed_strategy(strategy="collective")`, it would initialize Horovod and SOK related modules.

```
import tensorflow as tf
tf.config.experimental.enable_distributed_strategy(strategy="collective")
```

#### Embeddings

GroupEmbedding provide two level of API. The one is `tf.nn.group_embedding_lookup_sparse` and the other is `tf.feature_column.group_embedding_column_scope` whis is based on `feature_column` API.

#### group\_embedding\_lookup\_sparse

```
def group_embedding_lookup_sparse(params,
                                sp_ids,
                                combiners,
                                partition_strategy="mod",
                                sp_weights=None,
                                name=None):
```

- `params` : List, This parameter could receive one or more EmbeddingVariables or native Tensorflow Variable.
- `sp_ids` : List | Tuple , SparseTensor `sp_ids` is the ID used for EmbeddingLookup, the length must be consistent with `params`.

- `combiners` : List | Tuple, The pooling method of embedding values. Currently support mean and sum.
- `partition_strategy` : str, Currently not supported.
- `sp_weights` : List | Tuple the weight of `sp_ids` values. (Currently not supported.)
- `name` : str group name

### group\_embedding\_column\_scope

```
def group_embedding_column_scope(name=None):
```

- `name` : The name of scope.

We need to initialize a context `group_embedding_column_scope`, and complete the construction of `EmbeddingColumn` in that context. Later, the `EmbeddingColumn` Lookup would be simultaneously aggregate by `tf.feature_column.input_layer`. It is worth noting that the underlying implementation of this function is designed for `tf.RaggedTensor`. Although the IDS for `EmbeddingLookup` also supports `SparseTensor`, it will still be converted to `RaggedTensor` in the end, which will introduce a certain performance overhead.

## Original API in SOK

### Initialize

```
sparse_operation_kit.core.initialize.Init(**kwargs)
```

Abbreviated as `sok.Init(**kwargs)`.

This function is used to do the initialization of `SparseOperationKit` (SOK). In DeepRec, SOK supports Horovod. It can be used as:

```
sok_init = sok.Init(global_batch_size=args.global_batch_size)
with tf.Session() as sess:
    sess.run(sok_init)
...
```

## Embeddings

Embeddings includes sparse embedding and dense embedding. The `sok.DistributedEmbedding` is equivalent to `tf.nn.embedding_lookup_sparse` and `sok.All2AllDenseEmbedding` is equivalent to `tf.nn.embedding_lookup`.

### Distributed Sparse Embedding

```
class sparse_operation_kit.embeddings.distributed_embedding.
↳ DistributedEmbedding(combiner, max_vocabulary_size_per_gpu, embedding_vec_size, slot_
↳ num, max_nnz, max_feature_num=1, use_hashtable=True, **kwargs)
```

This is a wrapper class for distributed sparse embedding layer. It can be used to create a sparse embedding layer which will distribute keys based on `gpu_id = key % gpu_num` to each GPU. Parameters:

- `combiner` (string) – it is used to specify how to combine embedding vectors intra slots. Can be Mean or Sum.
- `max_vocabulary_size_per_gpu` (integer) – the first dimension of embedding variable whose shape is `[max_vocabulary_size_per_gpu, embedding_vec_size]`.

- `embedding_vec_size` (integer) – the second dimension of embedding variable whose shape is `[max_vocabulary_size_per_gpu, embedding_vec_size]`.
- `slot_num` (integer) – the number of feature-files which will be processed at the same time in each iteration, where all feature-files produce embedding vectors of the same dimension.
- `max_nnz` (integer) – the number of maximum valid keys in each slot (feature-filed).
- `max_feature_num` (integer = `slot_num*max_nnz`) – the maximum valid keys in each sample. It can be used to save GPU memory when this statistic is known. By default, it is equal to .
- `use_hashtable` (boolean = True) – whether using Hashtable in EmbeddingVariable, if True, Hashtable will be created for dynamic insertion. Otherwise, the input keys will be used as the index for embedding vector looking-up, so that input keys must be in the range `[0, max_vocabulary_size_per_gpu * gpu_num]`.
- `key_dtype` (`tf.dtypes = tf.int64`) – the data type of input keys. By default, it is `tf.int64`.
- `embedding_initializer` (string or an instance of `tf.keras.initializers.Initializer`) – the initializer used to generate initial value for embedding variable. By default, it will use `random_uniform` where `minval=-0.05`, `maxval=0.05`.

### All2All Dense Embedding

```
class sparse_operation_kit.embeddings.all2all_dense_embedding.All2AllDenseEmbedding(max_  
↪ vocabulary_size_per_gpu, embedding_vec_size, slot_num, nnz_per_slot, dynamic_  
↪ input=False, use_hashtable=True, **kwargs)
```

Abbreviated as `sok.All2AllDenseEmbedding(*args, **kwargs)`.

This is a wrapper class for all2all dense embedding layer. It can be used to create a dense embedding layer which will distribute keys based on `gpu_id = key % gpu_num` to each GPU. Parameters

- `max_vocabulary_size_per_gpu` (integer) – the first dimension of embedding variable whose shape is `[max_vocabulary_size_per_gpu, embedding_vec_size]`.
- `embedding_vec_size` (integer) – the second dimension of embedding variable whose shape is `[max_vocabulary_size_per_gpu, embedding_vec_size]`.
- `slot_num` (integer) – the number of feature-files which will be processed at the same time in each iteration, where all feature-files produce embedding vectors of the same dimension.
- `nnz_per_slot` (integer) – the number of valid keys in each slot. The number of valid keys in each slot is the same.
- `dynamic_input` (boolean = False) – whether the `inputs.shape` is dynamic. For example, the inputs tensor is coming from `tf.unique`. When `dynamic_input=True`, `unique->lookup->gather` pattern can be used. By default, it is False, which means the `inputs.size` must be `replica_batchsize * slot_num * nnz_per_slot`.
- `use_hashtable` (boolean = True) – whether using Hashtable in EmbeddingVariable, if True, Hashtable will be created for dynamic insertion. Otherwise, the input keys will be used as the index for embedding vector looking-up, so that input keys must be in the range `[0, max_vocabulary_size_per_gpu * gpu_num]`.
- `key_dtype` (`tf.dtypes = tf.int64`) – the data type of input keys. By default, it is `tf.int64`.
- `embedding_initializer` (string or an instance of `tf.keras.initializers.Initializer`) – the initializer used to generate initial value for embedding variable. By default, it will use `random_uniform` where `minval=-0.05`, `maxval=0.05`.

## Optimizers

The `unique` and `unsorted_segment_sum` are replaced with GPU implementations.

**Adam optimizer** `classsparse_operation_kit.tf.keras.optimizers.adam.Adam(*args, **kwargs)`

**Local update Adam optimizer** `classsparse_operation_kit.tf.keras.optimizers.lazy_adam.LazyAdamOptimizer(*args, **kwargs)`

## Utilizers

`sparse_operation_kit.optimizers.utils.split_embedding_variable_from_others(variables)` This function is used to split embedding variables from other variables.

Abbreviated as `sok.split_embedding_variable_from_others(variables)`.

Embedding variables are automatically created along with embedding layers. Since the aggregation for embedding variables is different from other variables, we need to split embedding variable and other variables so that optimizer can process those variables in different way.

Parameters

- `variables` (list, tuple) – a list or tuple of trainable `tf.Variable`.

Returns

- `embedding_variables` (tuple) – all embedding variables in the input variable-list.
- `other_variables` (tuple) – all normal variables in the input variable-list.

### 3.9.4 Detailed Doc

- For detailed introduction of GroupEmbedding, user can refer [GroupEmbedding documents](#)
- For detailed introduction, user can refer [SparseOperationKit documents](#).

## 3.10 Embedding Subgraph Fusion

### 3.10.1 Introduction

The native embedding lookup API's of DeepRec and TensorFlow, such as `safe_embedding_lookup_sparse`, create quite a lot of ops, and thus often run into kernel launch bound issues when executed on a GPU. To solve this problem, the Embedding Subgraph Fusion feature provides a set of APIs and fusion ops that can reduce the number of kernels to be launched. Together with high-performance implementations, it can accelerate execution on the GPU.

### 3.10.2 FeatureColumn API

Users interact with FeatureColumn as an interface. The `embedding_column` function returns an instance of the `EmbeddingColumn` class, commonly used `EmbeddingColumn` are:

1. `EmbeddingColumn` in `tensorflow/python/feature_column/feature_column_v2.py`
2. `_EmbeddingColumn` in `tensorflow/contrib/layers/python/layers/feature_column.py`

Then, it is typically passed into high level interfaces such as `tf.feature_column.input_layer` or `tf.feature_column_ops.input_from_feature_columns` to build the lookup-related computation graph.

Therefore, the Embedding Subgraph Fusion feature has added a `do_fusion` attribute to the above-mentioned `EmbeddingColumn` classes, which defaults to `False`. The user can set it to `True` explicitly when using it, making the embedding lookup process use fusion ops.

An examples:

```
import tensorflow as tf
from tensorflow.python.framework import ops

columns = tf.feature_column.categorical_column_with_embedding("col_emb", dtype=tf.dtypes.
    ↪int64)
W = tf.feature_column.embedding_column(categorical_column=columns,
    dimension=3,
    initializer=tf.ones_initializer(tf.dtypes.float32),
    do_fusion=True)

ids={}
ids["col_emb"] = tf.SparseTensor(indices=[[0,0],[1,1],[2,2],[3,3],[4,4]], values=tf.
    ↪cast([1,2,3,4,5], tf.dtypes.int64), dense_shape=[5, 4])

emb = tf.feature_column.input_layer(ids, [W])
fun = tf.multiply(emb, 2.0, name='multiply')
loss = tf.reduce_sum(fun, name='reduce_sum')
opt = tf.train.FtrlOptimizer(0.1, l1_regularization_strength=2.0, l2_regularization_
    ↪strength=0.00001)
g_v = opt.compute_gradients(loss)
train_op = opt.apply_gradients(g_v)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    print("init global done")
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
```

```
import tensorflow as tf
from tensorflow.python.framework import ops
from tensorflow.contrib.layers.python.layers import feature_column_ops
from tensorflow.contrib.layers.python.layers import feature_column
```

(continues on next page)



(continued from previous page)

```

columns = feature_column.sparse_column_with_embedding(column_name="col_emb", dtype=tf.
↳ dtypes.int64)
W = feature_column.embedding_column(sparse_id_column=columns,
    dimension=3,
    initializer=tf.ones_initializer(tf.dtypes.float32),
    do_fusion=True)

ids={}
ids["col_emb"] = tf.SparseTensor(indices=[[0,0],[1,1],[2,2],[3,3],[4,4]], values=tf.
↳ cast([1,2,3,4,5], tf.dtypes.int64), dense_shape=[5, 4])

# 传入设置了 do_fusion 的 EmbeddingColumn 实例
emb = feature_column_ops.input_from_feature_columns(columns_to_tensors=ids, feature_
↳ columns=[W])
fun = tf.multiply(emb, 2.0, name='multiply')
loss = tf.reduce_sum(fun, name='reduce_sum')
opt = tf.train.FtrlOptimizer(0.1, l1_regularization_strength=2.0, l2_regularization_
↳ strength=0.00001)
g_v = opt.compute_gradients(loss)
train_op = opt.apply_gradients(g_v)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    print("init global done")
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))

```

### 3.10.3 fused\_safe\_embedding\_lookup\_sparse API

Use `fused_safe_embedding_lookup_sparse` in `tf.contrib.layers.python.layers.embedding_ops` or `tf.python.ops.embedding_ops`

```

def fused_safe_embedding_lookup_sparse(embedding_weights,
    sparse_ids,
    sparse_weights=None,
    combiner="mean",
    default_id=None,
    name=None,
    partition_strategy="div",
    max_norm=None,
    prune=True):

```

This API is consistent with the functionality of DeepRec's `safe_embedding_lookup_sparse` interface. Therefore, the parameters will not be discussed again and can be viewed in the related documentation.

### 3.10.4 fused\_embedding\_lookup\_sparse API

Use `nn.fused_embedding_lookup_sparse`

```
def fused_embedding_lookup_sparse(params,
                                  sp_ids,
                                  sparse_weights=None,
                                  partition_strategy=None,
                                  name=None,
                                  combiner=None,
                                  max_norm=None,
                                  default_id=None,
                                  prune_invalid_ids=False,
                                  blocknums=None):
```

- `params`: List, which can contain a single embedding tensor or partitioned embedding tensors. The rank of embedding tensors must be 2.
- `sp_ids`: SparseTensor, whose values are the IDs to be looked up. The rank of the indices must be 2. The rank of the dense\_shape must be 1, and the number of elements must be 2.
- `sparse_weights`: weights of the values of the `sp_ids`.
- `partition_strategy`: The partition strategy of the embedding tensors.
- `name`: The name of this operation.
- `combiner`: The strategy to combine dimensions of entries.
- `max_norm`: If not None, calculate the l2 for each.
- `embedding`: vector, and normalize for values that exceed `max_norm`.
- `default_id`: For empty rows, fill in `default_id`. If `default_id` is None, fill in 0 by default.
- `prune_invalid_ids`: whether to remove invalid values (`id < 0`) from `sp_ids`
- `blocknums`: parameter used for DynamicEmbeddingVariable.

### 3.10.5 Please Note

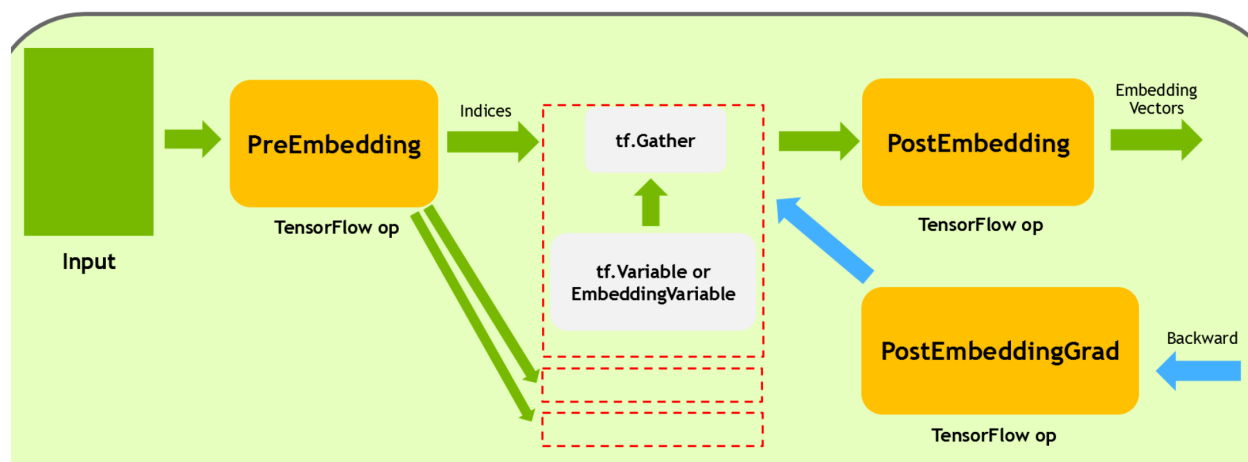
1. Currently, GPU Embedding subgraph Fusion only supports execution on Nvidia GPUs. The corresponding `tf.Variable` and `EmbeddingVariable`, as well as other operators, can run on the CPU.
2. Currently, setting weights `sparse_weights` is not supported.
3. The `partition_strategy` currently only supports `div`, and the embedding tensor is split on `axis = 0`. If the embedding tensor is an `EmbeddingVariable`, it can only be a single `ev` at present, and the partitioned lookup for `ev` is not supported.
4. Currently, the Dynamic Dimension, Multi-Hash Variable, and AdaptiveEmbedding features are not supported, and will be supported gradually in the future.

### 3.10.6 Op and computation graph

Newly added Embedding Fusion related ops:

1. FusedEmbeddingSparsePreLookUp
2. FusedEmbeddingSparsePostLookUp
3. FusedEmbeddingSparsePostLookUpGrad

Calling the low-level API `fused_embedding_lookup_sparse` will create the following computation graph:



1. **FusedEmbeddingSparsePreLookUp** is mainly responsible for filling empty rows, pruning invalid IDs, and partitioning the values and indices of `sp_ids` according to the `partition_strategy`.
2. **tf.Gather** is located on the same device as **EmbeddingVariable** or **tf.Variable** and in case of partition, there may be multiple copies of it, located on different devices (distributed). It receives the values and indices partitioned by **PreEmbedding**, and performs the actual embedding vector lookups.
3. **FusedEmbeddingSparsePostLookUp** then collects the embedding vectors from each partition, and performs related operations such as combiner and `max_norm`.
4. **FusedEmbeddingSparsePostLookUpGrad** is responsible for calculating the backward gradient of **FusedEmbeddingSparsePostLookUp**.

### 3.10.7 Performance

We compared performance of several models in modelzoo w/ and w/o fusion(average runtime of 5000 iterations):

Machine: 8 cores AMD EPYC 7232P CPU @ 3.20GHz.

A100-80GB-PCIE GPU

DLRM Model:

	Avg Time per Iteration
Unfused	20.78 ms
Fused	17.41 ms
SpeedUp	1.19x

DeepFM Model:

	Avg Time per Iteration
Unfused	37.24 ms
Fused	30.98 ms
SpeedUp	1.20x

WDL Model:

	Avg Time per Iteration
Unfused	36.38 ms
Fused	34.52 ms
SpeedUp	1.05x

## 3.11 Pipeline-Stage

### 3.11.1 Background

A TensorFlow training task is usually composed of sample data reading and graph calculation. The reading of sample data is an IO bound operation, which occupies a large percentage of the entire E2E time, then slowing down the training task. It cannot efficiently use computing resources (CPU, GPU). DeepRec has already provided the stage function. Its idea comes from the StagingArea function of TensorFlow. We provide the API `tf.staged` in DeepRec. The user explicitly specifies which part of the graph needs the stage, and adds `tf.make_prefetch_hook()` when the Session is created. Under the asynchronous execution of TensorFlow runtime, the execution efficiency of the entire graph is improved.

### 3.11.2 API

`tf.staged`, prefetch the input features, and return the prefetched tensor.

parameter	description	default value
features	The tensor that needs to be executed asynchronously: tensor, list of tensor (each element in the list is a tensor) or dict of tensor (the keys in the dict are all strings, and the values are all tensors).	required
feed_dict	A dictionary that maps graph elements to tensors.	{}
capacity	The maximum number of cached items asynchronous execution results.	1
num_threads	Number of threads to execute items asynchronously.	1
num_client	Number of clients of prefetched sample.	1
time-out_millis	Max milliseconds put op can take.	300000 ms
closed_exception_types	Exception types recognized as graceful exits.	(tf.errors.OUT_OF_RANGE,)
ignored_exception_types	Exception types that are recognized to be ignored and skipped.	()
use_stage_subgraph_thread_pool	Whether thread pool stage subgraph on an independent thread pool, you need to create an independent thread pool first.	False (If it is True, a separate thread pool must be created first)
stage_subgraph_thread_pool_index	If thread pool is stage subgraph to run on the independent thread pool to specify the independent thread pool index, you need to create an independent thread pool first, and enable the use_stage_subgraph_thread_pool option.	0, The index range is [0, the number of independent thread pools created - 1]
stage_subgraph_gpu_stream_index	In the GPU Multi-Stream scenario, the index of gpu stream used by stage subgraph.	0 (0 means that the stage subgraph shares the gpu stream used by the main graph, the index range is [0, total number of GPU streams - 1])
name	Name of prefetching operations.	None (Automatic generated)

Adds `tf.make_prefetch_hook()` hook when create session.

```
hooks=[tf.make_prefetch_hook()]
with tf.train.MonitoredTrainingSession(hooks=hooks, config=sess_config) as sess:
```

- Create a separate thread pool (optional)

```
sess_config = tf.ConfigProto()
sess_config.session_stage_subgraph_thread_pool.add() # Add a thread pool
sess_config.session_stage_subgraph_thread_pool[0].inter_op_threads_num = 8 # inter-
↪ thread number in thread pool
sess_config.session_stage_subgraph_thread_pool[0].intra_op_threads_num = 8 # intra-
↪ thread number in thread pool
sess_config.session_stage_subgraph_thread_pool[0].global_name = "StageThreadPool_1" #
↪ thread pool name
```

- GPU Multi-Stream Stage (optional)

```
sess_config = tf.ConfigProto()
sess_config.graph_options.rewrite_options.use_multi_stream = (rewriter_config_pb2.
↪ RewriterConfig.ON) # enable GPU Multi-Stream
sess_config.graph_options.rewrite_options.multi_stream_opts.multi_stream_num = 2 # The
↪ number of gpu streams, stream 0 is used by the main graph
sess_config.graph_options.optimizer_options.stage_multi_stream = True # enable GPU Multi-
↪ Stream Stage
```

GPU Multit-Stream Stage can achieve better performance by enabling GPU MPS, please refer to [GPU-MultiStream](#).

**Attention:**

- Computations to be asynchronous should compete with subsequent main computations for resources as little as possible (gpu, cpu, thread pool, etc.)
- A larger capacity will consume more memory or video memory, and may occupy CPU resources for subsequent model training. It is recommended to set it to follow-up calculation time/waiting for asynchronization time. It can be adjusted gradually upwards starting from 1.
- num\_threads is not as big as possible, it just needs to allow calculation and preprocessing to overlap, and a larger number will preempt CPU resources for model training. Calculation formula: num\_threads >= preprocessing time / training time, can be adjusted upwards from 1.
- tf.make\_prefetch\_hook() must be added, otherwise it will hang.

### 3.11.3 Example

```
import tensorflow as tf

filename_queue = tf.train.string_input_producer(['1.txt'])
reader = tf.TextLineReader()
k, v = reader.read(filename_queue)

var = tf.get_variable("var", shape=[100, 3], initializer=tf.ones_initializer())
v = tf.train.batch([v], batch_size=2, capacity=20 * 3)
v0, v1 = tf.decode_csv(v, record_defaults=[[''], ['']], field_delim=',')
xx = tf.staged([v0, v1])

xx[0]=tf.string_to_hash_bucket(xx[0],num_buckets=10)
xx[0] = tf.nn.embedding_lookup(var, xx[0])
xx[1]=tf.concat([xx[1], ['xxx']], axis = 0)
target = tf.concat([tf.as_string(xx[0]), [xx[1], xx[1]]], 0)

# mark target node
tf.train.mark_target_node([target])

with tf.train.MonitoredTrainingSession(hooks=[tf.make_prefetch_hook()]) as sess:
    for i in range(5):
        print(sess.run([target]))
```

## 3.12 Pipeline-SmartStage

### 3.12.1 Background

DeepRec provides the stage feature, which can realize the asynchronous execution of IO Bound operations and calculation Bound operations driven by TensorFlow runtime, improving the execution efficiency of the entire graph.

tf.staged requires the user to specify the boundary of the stage, on the one hand, it will increase the difficulty of use, on the other hand, the granularity of the stage will not be fine enough, making it difficult to execute more ops

asynchronously. So we propose the SmartStage feature. When users do not need to have an OP-level understanding of TF Graph, they can maximize the performance of the stage.

### 3.12.2 Feature

By enabling the smart stage feature, it automatically optimizes the maximum possible stage range from a certain starting node and modifies the actual physical calculation graph (without affecting the Graphdef), improving performance.

### 3.12.3 API

#### 1. Automatically SmartStage (Recommend)

The premise of automatic SmartStage is that the model uses the `tf.data.Iterator` interface to read sample data from `tf.data.Dataset`.

1. The `tf.SmartStageOptions` interface returns the configuration for executing the stage subgraph, and its parameters are as follows:

parameter	description	default value
capacity	The maximum number of cached asynchronous execution results.	1
num_threads	Number of threads to execute stage subgraph asynchronously.	1
num_client	Number of clients of prefetched sample.	1
time-out_millis	Max milliseconds put op can take.	300000 ms
closed_exception_types	Exception types recognized as graceful exits.	( <code>tf.errors.OUT_OF_RANGE</code> ,)
ignored_exception_types	Exception types that are recognized to be ignored and skipped.	()
use_stage_subgraph_thread_pool	Whether to use the stage subgraph on an independent thread pool, you need to create an independent thread pool first.	False (If it is True, a separate thread pool must be created first)
stage_subgraph_thread_pool_index	Apply thread pool to the stage subgraph to run on the independent thread pool to specify the independent thread pool index, you need to create an independent thread pool first, and enable the <code>use_stage_subgraph_thread_pool</code> option.	0, The index range is [0, the number of independent thread pools created - 1]
stage_subgraph_gpu_stream_index	In the GPU Multi-Stream scenario, the index of gpu stream used by stage subgraph.	0 (0 means that the stage subgraph shares the gpu stream used by the main graph, the index range is [0, total number of GPU streams - 1])
graph	The Graph that needs to be optimized by SmartStage, which is the same as the Graph passed to the Session	None (Use default graph)
name	Name of prefetching operations.	None (Automatic generated)

For how to create an independent thread pool or use GPU Multi-Stream, please refer to [Pipeline-Stage](#).

2. The configuration generated by the `tf.SmartStageOptions` interface needs to be assigned to `tf.ConfigProto`.

```
sess_config = tf.ConfigProto()
smart_stage_options = tf.SmartStageOptions(capacity=40, num_threads=4)
sess_config.graph_options.optimizer_options.smart_stage_options.CopyFrom(smart_
↪stage_options)
```

3. Set the following options in `tf.ConfigProto` to enable SmartStage.

- CPU scenario

```
sess_config = tf.ConfigProto()
sess_config.graph_options.optimizer_options.do_smart_stage = True
```

- GPU scenario

```
sess_config = tf.ConfigProto()
sess_config.graph_options.optimizer_options.do_smart_stage = True
sess_config.graph_options.optimizer_options.stage_subgraph_on_cpu = True
```

4. Add `tf.make_prefetch_hook()` hook to Session.

## 2. SmartStage when Graph contains Stage

The original graph has been manually split using the `tf.staged` interface.

For more detail of `tf.staged`, please refer to [Pipeline-Stage](#).

1. Set the following options in `tf.ConfigProto` to enable SmartStage.

- CPU scenario

```
sess_config = tf.ConfigProto()
sess_config.graph_options.optimizer_options.do_smart_stage = True
```

- GPU scenario

```
sess_config = tf.ConfigProto()
sess_config.graph_options.optimizer_options.do_smart_stage = True
sess_config.graph_options.optimizer_options.stage_subgraph_on_cpu = True
```

2. Add `tf.make_prefetch_hook()` hook to Session.

### 3.12.4 Example

#### Automatically SmartStage (Recommend)

```
import tensorflow as tf

def parse_csv(value):
    v = tf.io.decode_csv(value, record_defaults=[[''], ['']])
    return v

dataset = tf.data.TextLineDataset('./test_data.csv')
dataset = dataset.batch(2)
dataset = dataset.map(parse_csv, num_parallel_calls=2)
dataset_output_types = tf.data.get_output_types(dataset)
dataset_output_shapes = tf.data.get_output_shapes(dataset)
iterator = tf.data.Iterator.from_structure(dataset_output_types, dataset_output_shapes)
xx = iterator.get_next()
xx = list(xx)
```

(continues on next page)



(continued from previous page)

```

init_op = iterator.make_initializer(dataset)

var = tf.get_variable("var", shape=[100, 3], initializer=tf.ones_initializer())
xx[0] = tf.string_to_hash_bucket(xx[0], num_buckets=10)
xx[0] = tf.nn.embedding_lookup(var, xx[0])
xx[1]=tf.concat([xx[1], ['xxx']], axis = 0)
target = tf.concat([tf.as_string(xx[0]), [xx[1], xx[1]]], 0)

config = tf.ConfigProto()
# enable smart stage
config.graph_options.optimizer_options.do_smart_stage = True
smart_stage_options = tf.SmartStageOptions(capacity=1, num_threads=1)
config.graph_options.optimizer_options.smart_stage_options.CopyFrom(smart_stage_options)

# For GPU training, consider enabling the following options for better performance
# config.graph_options.optimizer_options.stage_subgraph_on_cpu = True

# mark target 节点
tf.train.mark_target_node([target])

scaffold = tf.train.Scaffold(
    local_init_op=tf.group(tf.local_variables_initializer(), init_op))
with tf.train.MonitoredTrainingSession(config=config, scaffold=scaffold,
                                       hooks=[tf.make_prefetch_hook()]) as sess:
    for i in range(5):
        print(sess.run([target]))

```

### SmartStage when Graph contains Stage.

```

import tensorflow as tf

filename_queue = tf.train.string_input_producer(['1.txt'])
reader = tf.TextLineReader()
k, v = reader.read(filename_queue)

var = tf.get_variable("var", shape=[100, 3], initializer=tf.ones_initializer())
v = tf.train.batch([v], batch_size=2, capacity=20 * 3)
v0, v1 = tf.decode_csv(v, record_defaults=[[''], ['']], field_delim=',')
xx = tf.staged([v0, v1])

xx[0]=tf.string_to_hash_bucket(xx[0],num_buckets=10)
xx[0] = tf.nn.embedding_lookup(var, xx[0])
xx[1]=tf.concat([xx[1], ['xxx']], axis = 0)
target = tf.concat([tf.as_string(xx[0]), [xx[1], xx[1]]], 0)

config = tf.ConfigProto()
# enable smart stage
config.graph_options.optimizer_options.do_smart_stage = True
# For GPU training, consider enabling the following options for better performance

```

(continues on next page)

(continued from previous page)

```
# config.graph_options.optimizer_options.stage_subgraph_on_cpu = True

# mark target node
tf.train.mark_target_node([target])

with tf.train.MonitoredTrainingSession(config=config,
                                       hooks=[tf.make_prefetch_hook()]) as sess:
    for i in range(5):
        print(sess.run([target]))
```

### 3.12.5 Performance

- CPU scenario

The performance of this feature in the DLRM model in modelzoo.

The Aliyun ECS instance is ecs.hfg7.8xlarge

- Model name: Intel(R) Xeon(R) Platinum 8369HC CPU @ 3.30GHz
- CPU(s): 32
- Socket(s): 1
- Core(s) per socket: 16
- Thread(s) per core: 2
- Memory: 128G

	case	global steps/sec
DLRM	w/o smart stage	201 (baseline)
DLRM	w/ smart stage	212 (+ 1.05x)

- GPU scenario

The performance of this feature in the GPU training scenario in the model in modelzoo.

machine configuration:

Resource	Description	Cores
CPU	Intel(R) Xeon(R) Platinum 8369B CPU @ 2.90GHz	64 core
GPU	NVIDIA A100 80G	1 gpu
MEM	492G	

Performance results comparison:

model	w/o SmartStage (global steps/sec)	do_smartstage (global steps/sec)	do_smartstage_gpu (global steps/sec)
DIEN	17.1673	16.918	17.2557
DIN	137.584	132.619	165.069
DLRM	91.6982	67.735	188.105
DSSM	92.4544	83.7194	101.352
DeepFM	74.7011	62.1227	93.0858

## 3.13 Asynchronous Embedding Lookup

### 3.13.1 Background

In the scenario of distributed training of sparse models, as the model becomes more and more complex, the input features of the model also increase. As a result, the worker node needs to perform a large number of embedding lookup operations on the ps node during each step of training, causing the time-consuming proportion of this operation to increase in training, which becomes the bottleneck of the model training speed.

DeepRec provides the function of asynchronous Embedding lookup, which can automatically determine the subgraph of Embedding lookup and realize the asynchronous execution with the main graph, thereby eliminating the impact of model communication bottleneck on training and improving training performance.

After the embedding lookup operation is asynchronous, the gradient update operation and the embedding lookup operation of the work node are located in different graphs. As a result, the worker node could not obtain the latest embedding lookup result from the PS node, which may affect the training convergence speed and model effect.

### 3.13.2 Description

On the premise that the user's original graph has an io stage, the Asynchronous Embedding Lookup function can automatically determine the subgraph of Embedding lookup and realize the asynchronous execution with the main graph.

#### Attention

1. The prerequisite for enabling the Asynchronous Embedding Lookup function is that there is an io stage in the user's original graph, which should be after the sample reading and before the embedding lookup operation. Please refer to *Pipeline-Stage*.
2. This function conflicts with *Pipline-SmartStage*, and the SmartStage function will be turned off if the Asynchronous Embedding Lookup function is enabled.

### 3.13.3 User API

Currently, the Asynchronous Embedding Lookup function supports the following Embedding Lookup interfaces in DeepRec.

```
tf.contrib.feature_column.sequence_input_layer()
tf.contrib.layers.safe_embedding_lookup_sparse()
tf.contrib.layers.input_from_feature_columns()
tf.contrib.layers.sequence_input_from_feature_columns()
tf.feature_column.input_layer()
tf.nn.embedding_lookup()
tf.nn.embedding_lookup_sparse()
tf.nn.safe_embedding_lookup_sparse()
tf.nn.fused_embedding_lookup_sparse()
tf.python.ops.embedding_ops.fused_safe_embedding_lookup_sparse()
```

DeepRec provides the following configuration options in ConfigProto:

```
sess_config = tf.ConfigProto()
sess_config.graph_options.optimizer_options.do_async_embedding = True
sess_config.graph_options.optimizer_options.async_embedding_options.threads_num = 4
```

(continues on next page)

(continued from previous page)

```

sess_config.graph_options.optimizer_options.async_embedding_options.capacity = 4
sess_config.graph_options.optimizer_options.async_embedding_options.use_stage_subgraph_
↪thread_pool = False # optional
sess_config.graph_options.optimizer_options.async_embedding_options.stage_subgraph_
↪thread_pool_id = 0 # optional

```

Configuration Options	Description	Default Value
do_async_embedding_lookup	Enable Asynchronous Embedding Lookup or not	False
threads_num	The number of threads that execute embedding lookup subgraph asynchronously	0 (must be set)
capacity	The maximum number of Asynchronous Embedding lookup results that a worker node can cache	0 (must be set)
use_stage_subgraph_thread_pool	Use an independent thread pool to run the embedding lookup subgraph or not, need to create an independent thread pool first	False(optional)
stage_subgraph_thread_pool_id	Index of independent thread pool	0(optional, the index range is [0, the number of independent thread pools - 1])

### Attention

1. `async_embedding_threads_num` is not as big as possible, you only need to let the execution of the main graph not have to wait for the result of the embedding lookup operation. If the value is too large, it will preempt computing resources for model training and occupy more communication bandwidth. It is recommended to set according to the following formula. You can also start at 1 and adjust upwards to find the best value.

$$async\_embedding\_threads\_num \geq \lceil T_{Embedding\_lookup} / T_{main\_graph} \rceil$$

2. A larger capacity will consume more memory, and will also cause a larger difference between the cached embedding lookup result and the latest result obtained from the PS node, resulting in slow training convergence. It is recommended to set it to the same value as `async_embedding_threads_num`, which can be adjusted upwards from 1.
3. The independent thread pool option can make different Stage subgraphs run in different thread pools, avoiding competition with the default thread pool for the main graph and other subgraphs. For how to create an independent thread pool, please refer to [Pipeline-Stage](#).

## 3.13.4 Performance

### CPU scenario

The performance of this feature in the DLRM model in modelzoo.

The Aliyun ECS instance is ecs.hfc7.24xlarge, the training cluster consists of 10 Aliyun ECS instances.

**Aliyun ECS Instance configuration information:**

	Description
CPU	Intel Xeon Platinum (Cooper Lake) 8369 96 cores
MEM	192 GiB
NET	32 Gbps

**Training configuration:**

	size
PS nodes	8
worker nodes	30
cpu cores per PS	15
cpu cores per worker	10

**Result**

model	baseline (global steps/sec)	async embedding (global steps/sec)	speedup
DLRM	1008.6968	1197.932	1.1876

## 3.14 Sample-awared Graph Compression

### 3.14.1 Background

In the recommendation scenario, the composition of the sample is often <user, item>, and it is a one-to-many composition form. It has the following characteristics:

#### Sample Features

As the source of model training, samples determine the composition of the model and affect the calculation of the model. In the recommendation scenario, several samples produced by one page exposure have the commonality of the same user, that is,

```

user_1_feature, item_1_feature, label_1
user_1_feature, item_2_feature, label_2
... ..
user_1_feature, item_N_feature, label_N

```

It leads to redundancy in the storage of samples, wastes storage space.

## Model Features

Under such sample characteristics, the redundant samples read in must also lead to computational redundancy. For the above samples, when training the model, inside the batch, `user_1_feature` will repeat the run sub-graph (such as: Attention Graph) several times. This part of the calculation is completely redundant and can be saved at runtime, only calculated once.

### 3.14.2 Feature

In the training and inference scenarios, using the characteristics of samples and models, without the need for users to modify the Graph, connect the compressed samples prepared by the user, automatically optimize the graph, and improve the performance of training and inference.

## Training & Inference

Before the user constructs the training/inference graph, call the python API to open the function and automatically optimize the graph.

### 3.14.3 API

```
tf.graph_optimizer.enable_sample_aware_graph_compression(user_tensors,
                                                         item_tensors,
                                                         item_size)
```

Args:

```
user_tensors: user sample
item_tensors: non-user sample
item_size: the number of items
```

### 3.14.4 Example

#### Training

TODO

#### Inference

```
USER_FEATURE = ['user_feature_0', 'user_feature_1']
ITEM_FEATURE = ['item_feature_0', 'item_feature_1']
ALL_FEATURE = USER_FEATURE + ITEM_FEATURE

def serving_input_receiver_fn():
    item_size = tf.placeholder(dtype=tf.int32, shape=[None], name='item_size')
    features = {}
    inputs = {"item_size": item_size}
    user_tensors = []
    item_tensors = []
    for fea_name in ALL_FEATURE:
        features[fea_name] = tf.placeholder(tf.string, [None], name=fea_name)
```

(continues on next page)

(continued from previous page)

```

inputs[fea_name] = features[fea_name]
if fea_name in ITEM_FEATURE:
    item_tensors.append(features[fea_name])
else:
    user_tensors.append(features[fea_name])

"""Enable Sample-awared Graph Compression"""
tf.graph_optimizer.enable_sample_aware_graph_compression(
    user_tensors,
    item_tensors,
    item_size)

return tf.estimator.export.ServingInputReceiver(features, inputs)

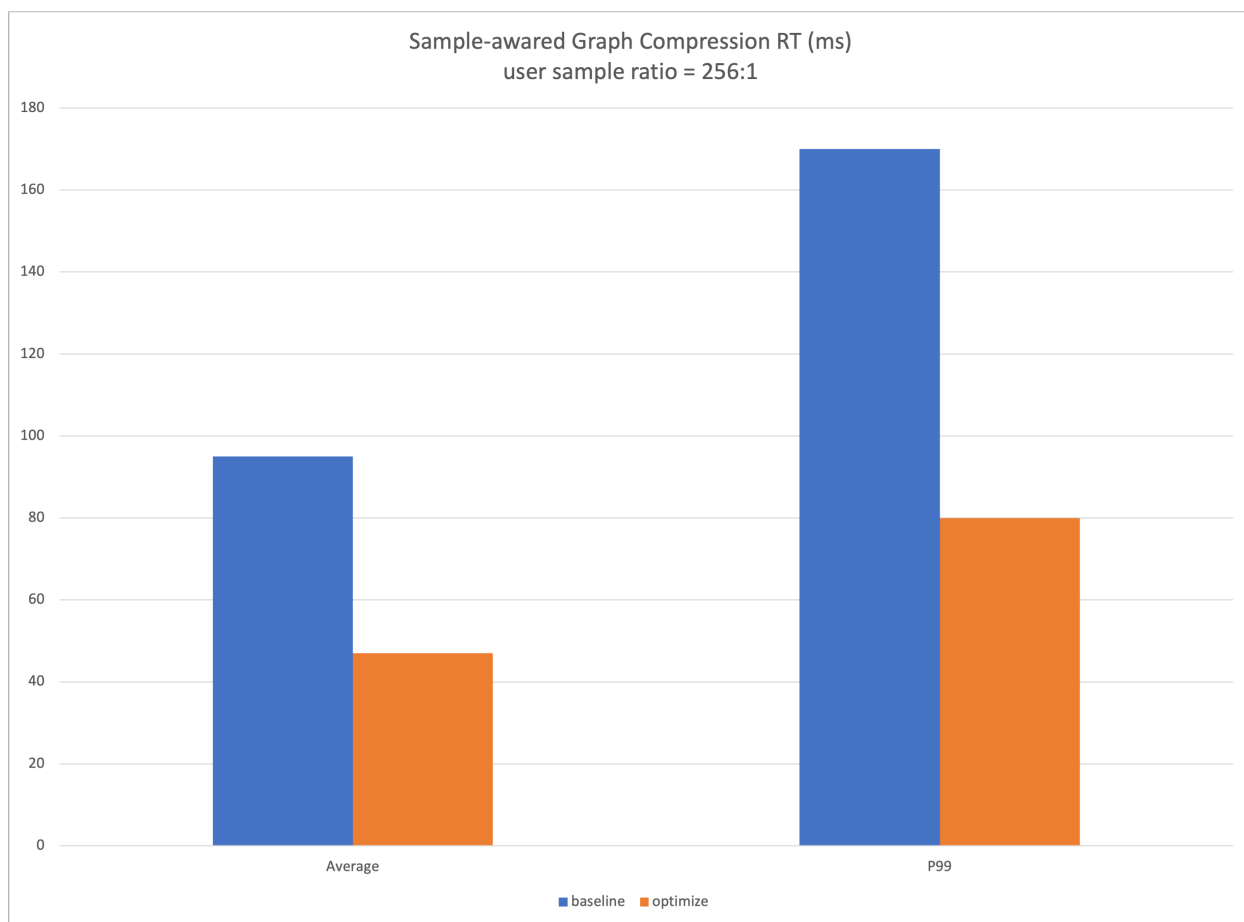
estimator = ...
estimator.export_savedmodel(output_dir, serving_input_receiver_fn)

```

1. Need to use `estimator` provided by DeepRec at the same time
2. Input data format Generally, in the inference scenario, the input data is protobuf, which contains the values of each feature required by the model and the tensor indicating the number of items in the sample. The user tensor shape is [Duser], and the item tensor shape is [N, Ditem]

### 3.14.5 Performance

Compressing user-side features reduces the end-to-end delay of Inference. In a cloud online service case, the performance results are as follows:



## 3.15 CPU Memory Optimization

### 3.15.1 Introduction

On the CPU side, memory libraries `ptmalloc` or `jemalloc` can cause severe page faults while allocating large memory chunks common to DL applications. To solve this issue, DeepRec optimizes memory allocation to reduce the memory usage and minor page faults, and improve the running performance. When this optimization is enabled, DeepRec will collect memory allocation information (after the number of steps reaches the `START_STATISTIC_STEP` threshold), and then generate an allocation plan based on the collected memory allocation information of each step. When generating the allocation plan, it will determine whether the previously generated memory allocation plan meets the current allocation requirements, and if it is considered a stable step. When the number of stable steps reaches the `STABLE_STATISTIC_STEP` threshold or the total number of steps collected reaches the `MAX_STATISTIC_STEP` threshold, DeepRec will stop collecting memory information. Since memory allocation information needs to be collected for optimization, the performance gain can only be observed after a certain number of steps.



### 3.15.2 User API

On the CPU side, the current version of DeepRec supports the CPU memory optimization of stand-alone and distributed training/inference, which is enabled by default, and can be turned off using the `export ENABLE_MEMORY_OPTIMIZATION=0` command. There are several environment variables. `START_STATISTIC_STEP` configures the step to start collecting memory information. `STABLE_STATISTIC_STEP` configures how many stable steps the allocation policy ends. `MAX_STATISTIC_STEP` configures the maximal steps to end the memory allocation policy. The default values are 100, 10, and 100, respectively. These values generally do not need to be changed, and the `START_STATISTIC_STEP` can be increased when there are many initialization graphs, and the `STABLE_STATISTIC_STEP` and `MAX_STATISTIC_STEP` can be increased when the main computational graph is irregular or there are more running computational graphs.

#### Using jemalloc

The CPU side can adapt the memory optimization with the jemalloc library. After setting the `MALLOC` environment variable, add the `LD_PRELOAD` jemalloc dynamic library before the python command, for example:

```
export MALLOC_CONF="background_thread:true,metadata_thp:auto,dirty_decay_ms:20000,muzzy_
↳decay_ms:20000"
LD_PRELOAD=./libjemalloc.so.2 python ...
```

## 3.16 GPU Memory Optimization

### 3.16.1 Introduction

On the GPU side, `BFCAllocator`, the GPU allocator of TensorFlow, has the problem of excessive memory fragmentation, resulting in wasted GPU memory. This optimization reduces memory fragmentation and results in a smaller memory footprint than `BFCAllocator`. This optimization collects memory allocation information and uses `cudaMalloc` to allocate GPU memory in the first K steps, so the performance will be degraded. After the information is collected, the optimization is enabled, and the performance will be improved.

### 3.16.2 User API

On the GPU side, the current version of DeepRec only supports the memory optimization of the stand-alone training. This optimization is turned off by default and can be turned on using the `export TF_GPU_ALLOCATOR=tenorpool` command. Note that this optimization currently performs slightly worse than `BFCAllocator`, but uses less memory than `BFCAllocator`. For example, the execution performance of the recommended model `DBMTL-MMOE` will decrease by about 1% (from 4.72 global\_step/sec to 4.67 global\_step/sec), but the GPU memory will be reduced by 23% (from 1509.12 MB to 1155.12 MB). There are two environment variables: `START_STATISTIC_STEP` and `STOP_STATISTIC_STEP` to configure the step to start collecting memory allocation information and the step to end collection and start optimizing, respectively. The default value are 10 and 110, respectively, which can be adjusted appropriately.

## 3.17 GPU Virtual Memory Optimization

### 3.17.1 Introduction

Training with GPUs often encounters the problem of insufficient GPU memory, this optimization uses cuda's `cuMemAllocManaged` API of the uniform memory address to allocate GPU memory when the GPU memory is not enough. This API uses CPU memory to increase GPU memory usage, note that this causes performance degradation.

### 3.17.2 User API

This optimization is enabled by default and is only used when there is insufficient GPU memory, and can be turned off using `export TF_GPU_VMEM=0`. Note that the current GPU Memory Optimization is not compatible with GPU Virtual Memory Optimization, and GPU Virtual Memory Optimization will be turned off when GPU Memory Optimization is enabled.

## 3.18 Executor Optimization

### 3.18.1 Introduction

DeepRec introduces some optimizations in runtime scheduling, currently mainly optimizing the execution efficiency of the CPU side, and will continue to launch GPU runtime optimization.

### 3.18.2 User API

Three executor policies are currently supported.

#### Native Tensorflow Executor

The default executor policy.

#### Inline Executor

This executor supports the session to execute on one thread, which reduces thread switching and supports high throughput under high concurrency, and reduce the scheduling overhead brought by the framework. It can generally be used in serving high concurrency scenarios and some scenarios that use parameter servers, which can bring good performance.

#### Usage

```
sess_config = tf.ConfigProto()
sess_config.executor_policy = tf.ExecutorPolicy.USE_INLINE_EXECUTOR

with tf.train.MonitoredTrainingSession(
    master=server.target,
    ...
    config=sess_config) as sess:
    ...
```

## CostModel-based Executor

This executor traces and collects the execution information of the model to build the CostModel and performs optimal execution scheduling based on the CostModel. This executor includes the critical path scheduling and a scheduling strategy for batching execution of operators with short time.

### Usage

Users can specify which steps to collect execution information. The default is to collect information in 100 to 200 steps. Users can customize both parameters by setting the following environment variables:

```
os.environ['START_NODE_STATS_STEP'] = "200"
os.environ['STOP_NODE_STATS_STEP'] = "500"
```

The above example represents to collect information in 100 to 200 steps. If the START\_NODE\_STATS\_STEP is less than STOP\_NODE\_STATS\_STEP, this executor will be disabled. At the same time, in the user script, the following code needs to be added to enable the CostModel-based Executor.

```
sess_config = tf.ConfigProto()
sess_config.executor_policy = tf.ExecutorPolicy.USE_COST_MODEL_EXECUTOR

with tf.train.MonitoredTrainingSession(
    master=server.target,
    ...
    config=sess_config) as sess:
    ...
```

## 3.19 GPU Multi-stream

### 3.19.1 Introduction

In training scenarios, GPUs are often used to accelerate computation. Since different computing kernels are only committed on the same stream, insufficient execution concurrency and low GPU utilization may occur under some models. To this end, we provide GPU Multi-stream optimization.

This feature provides multiple GPU streams, and has a variety of built-in graph splitting rules, and users can also manually specify the sub-graph. This feature enables several subgraphs without data dependency to be submitted to different GPU streams for execution, achieving concurrent execution at the subgraph level, thereby improving GPU utilization.

### 3.19.2 User API

This feature can be enabled by setting the following parameters in tf.ConfigProto.

```
import tensorflow as tf
from tensorflow.core.protobuf import rewriter_config_pb2
sess_config = tf.ConfigProto()
sess_config.graph_options.rewrite_options.use_multi_stream = (rewriter_config_pb2.
    ↪ RewriterConfig.ON) # Turn on the multi-stream feature
sess_config.graph_options.rewrite_options.multi_stream_opts.multi_stream_num = 4 # The
    ↪ number of streams
```

### 3.19.3 Graph Splitting Strategy

#### 1. Manual Graph Splitting

For manual graph splitting, we provide the `tf.stream()` API to specify the stream id, which can be nested.

At the same time, the `tf.colocate_with()` API also supports the requirement of associating the newly created operation with one specified operation and placing them on the same stream.

#### Usage

```
with tf.stream(0):
    # Set the context of stream 0, and the stream id should be less than the number of
    # streams specified in tf.ConfigProto
    a = tf.placeholder(tf.float32, [None, 1], name='a') # This operation will be placed on
    # stream 0
    with tf.stream(1):
        # Set the context of stream 1, and the stream id should be less than the number of
        # streams specified in tf.ConfigProto
        b = tf.placeholder(tf.float32, [None, 1], name='b') # This operation will be placed
        # on stream 1
        # Go back to the context of stream 0
        c = tf.constant([1, 2, 3, 4], tf.float32, [4, 1], name='c') # This operation will be
        # placed on stream 0

with tf.colocate_with(a):
    # Associated with `a`
    d = tf.constant([5, 6, 7, 8], tf.float32, [4, 1], name='d') # This operation is
    # associated with `a`, and will be placed on the same GPU stream of `a`
```

#### Best Practise

```
import tensorflow as tf
from tensorflow.core.protobuf import rewriter_config_pb2
import numpy as np
import os
os.environ['CUDA_VISIBLE_DEVICES']='0'

learning_rate = 0.01
max_train_steps = 1000
log_step = 100

train_X = np.array([[3.3], [4.4], [5.5], [6.71], [6.93], [4.168], [9.779],
                    [6.182], [7.59], [2.167], [7.042], [10.791], [5.313], [7.997],
                    [5.654], [9.27], [3.1]], dtype=np.float32)
train_Y = np.array([[3.3], [4.4], [5.5], [6.71], [6.93], [4.168], [9.779],
                    [6.182], [7.59], [2.167], [7.042], [10.791], [5.313], [7.997],
                    [5.654], [9.27], [3.1]], dtype=np.float32)
train_Z = np.array([[1.7], [2.76], [2.09], [3.19], [1.694], [1.573], [3.336],
                    [2.596], [2.53], [1.221], [2.827], [3.465], [1.65], [2.904],
```

(continues on next page)

(continued from previous page)

```

        [2.42], [2.94], [1.3]], dtype=np.float32)

total_samples = train_X.shape[0]

Z_ = tf.placeholder(tf.float32, [None, 1])

with tf.stream(0):
    X = tf.placeholder(tf.float32, [None, 1])
    W_X = tf.Variable(tf.random_normal([1, 1]), name='weight_x')
    b = tf.Variable(tf.zeros([1]), name='bias')
    X_Result = tf.matmul(X, W_X)
    X_Result = tf.add(X_Result, b)

with tf.stream(1):
    Y = tf.placeholder(tf.float32, [None, 1])
    W_Y = tf.Variable(tf.random_normal([1, 1]), name='weight_y')
    Y_Result = tf.matmul(Y, W_Y)

Z = X_Result + Y_Result
loss = tf.reduce_sum(tf.pow(Z-Z_, 2)) / (total_samples)
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
train_op = optimizer.minimize(loss)

sess_config = tf.ConfigProto()
sess_config.graph_options.rewrite_options.use_multi_stream = (rewriter_config_pb2.
↳RewriterConfig.ON)
sess_config.graph_options.rewrite_options.multi_stream_opts.multi_stream_num = 2
with tf.Session(config=sess_config) as sess:
    sess.run(tf.global_variables_initializer())
    print("Start training:")
    for step in range(max_train_steps):
        sess.run(train_op, feed_dict={X: train_X, Y: train_Y, Z_: train_Z})
        if step % log_step == 0:
            c = sess.run(loss, feed_dict={X: train_X, Y: train_Y, Z_: train_Z})
            print("Step:%d, loss==%.4f, W_X==%.4f, b==%.4f, W_Y==%.4f" %
                  (step, c, sess.run(W_X), sess.run(b), sess.run(W_Y)))
    final_loss = sess.run(loss, feed_dict={X: train_X, Y: train_Y, Z_: train_Z})
    w_x, b, w_y = sess.run([W_X, b, W_Y])
    print("Step:%d, loss==%.4f, W_X==%.4f, b==%.4f, W_Y==%.4f" %
          (max_train_steps, final_loss, w_x, b, w_y))
    print("Linear Regression Model: Z==%.4f*X + %.4f*Y + %.4f" % (w_x, w_y, b))

```

### 3.19.4 Enabling GPU MPS

This optimization can adapt GPU MPS (Multi-Process Service). Users can enable GPU MPS by following these steps.

1. The host starts the GPU MPS.

```
nvidia-cuda-mps-control -d
```

2. Docker launch configuration (if training inside the container)

The `--ipc=host` option needs to be added so that the GPU MPS can be communicated with the process in the container. The following is an example.

```
sudo docker run -itd --name <container_name> --ipc=host --gpus='"device=0"' <image_
↪id> bash
```

In this example, GPU 0 is bound to the created container, and one GPU is visible in the container. The GPU MPS can be used by directly executing GPU training tasks in the container.

## 3.20 Incremental Checkpoint

### 3.20.1 Introduction

In large-scale sparse training, the data is skewed, and most of the data in adjacent full checkpoints remains unchanged. In this context, saving only incremental checkpoints for sparse parameters will greatly reduce the overhead caused by frequently saving checkpoints. After the PS failover, try to restore the model parameters of the latest training on the PS through a recent full checkpoint and a series of incremental checkpoints to reduce repeated calculations.

### 3.20.2 API

```
def tf.train.MonitoredTrainingSession(..., save_incremental_checkpoint_secs=None, ...):
    pass
```

extra parameters:

`save_incremental_checkpoint_secs`, default: None. User can set the incremental\_save checkpoint time in seconds, to generate the incremental checkpoint.

### 3.20.3 Example

High-level API (`tf.train.MonitoredTrainingSession`)

```
import tensorflow as tf
import time, os

tf.logging.set_verbosity(tf.logging.INFO)

sparse_var=tf.get_variable("a", shape=[30,4], initializer=tf.ones_initializer(tf.
↪float32),partitioner=tf.fixed_size_partitioner(num_shards=4))
dense_var=tf.get_variable("b", shape=[30,4], initializer=tf.ones_initializer(tf.float32),
↪partitioner=tf.fixed_size_partitioner(num_shards=4))
```

(continues on next page)

(continued from previous page)

```

ids=tf.placeholder(dtype=tf.int64, name='ids')
emb = tf.nn.embedding_lookup(sparse_var, ids)

fun = tf.multiply(emb, 2.0, name='multiply')
loss = tf.reduce_sum(fun, name='reduce_sum')

gs = tf.train.get_or_create_global_step()

opt=tf.train.AdagradOptimizer(0.1, initial_accumulator_value=1)
g_v = opt.compute_gradients(loss)
train_op = opt.apply_gradients(g_v, global_step=gs)

path = 'test/test4tolerance/ckpt/'
with tf.train.MonitoredTrainingSession(checkpoint_dir=path,
    save_checkpoint_secs=60,
    save_incremental_checkpoint_secs=20) as sess:
    for i in range(1000):
        print(sess.run([gs, train_op, loss], feed_dict={"ids:0": i%10}))
        time.sleep(1)

```

Estimator

Configure parameters when constructing EstimatorSpec

tf.train.Saver and tf.train.Scaffold set incremental\_save\_restore=True, tf.train.CheckpointSaverHook set save incremental checkpoint interval incremental\_save\_secs

```

def model_fn(self, features, labels, mode, params):
    ...

    scaffold = tf.train.Scaffold(
        saver=tf.train.Saver(
            sharded=True,
            incremental_save_restore=True),
        incremental_save_restore=True)

    ...

    return tf.estimator.EstimatorSpec(
        mode,
        loss=loss,
        train_op=train_op,
        training_hooks=[logging_hook],
        training_chief_hooks=[
            tf.train.CheckpointSaverHook(
                checkpoint_dir=params['model_dir'],
                save_secs=params['save_checkpoints_secs'],
                save_steps=params['save_checkpoints_steps'],
                scaffold=scaffold,
                incremental_save_secs=120)],
        scaffold=scaffold)

```

### 3.20.4 Model Export

By default, incremental checkpoint subgraphs cannot be exported to SavedModel. If users want to support second-level updates through “incremental model update” in Serving, they need to export incremental checkpoint subgraphs to SavedModel. You need to use the [Estimator](#) provided by DeepRec to export incremental checkpoint subgraphs.

Example:

```
estimator.export_saved_model(  
    export_dir_base,  
    serving_input_receiver_fn,  
    ...  
    save_incr_model=True)
```

Attention:

When there is no incremental model when building graph, an error will be reported when configuring `save_incr_model=True`, so there is only the full amount in the graph, and `save_incr_model` can only be configured with false (default value). When there are full and incremental models in the graph, `save_incr_model` is set to true, and the SavedModel graph can load full or incremental models. If `save_incr_model` is set to false, the SavedModel graph can only load the full model.

## 3.21 Embedding Variable Export Format

### 3.21.1 Introduction

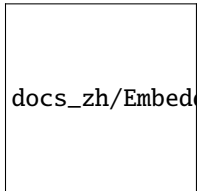
EmbeddingVariable is the expression of sparse parameters in DeepRec. After training the model, the user needs to obtain sparse parameter information from checkpoint, including: sparse features, feature frequency, and feature versions.

EmbeddingVariable is stored in ckpt in the form of tensor, assuming the name of the EV in the graph is `a/b/c/embedding`, then there are four tensors in checkpoint:

1. “a/b/c/embedding-keys” Tensor stores EV keys, `shape=[N]`, `dtype=int64`;
2. “a/b/c/embedding-values” Tensor stores EV value, `shape=[N, embedding_dim]`, `dtype=float`;
3. “a/b/c/embedding-freqs” Tensor stores EV frequency, `shape=[N]`, `dtype=int64`;
4. “a/b/c/embedding-versions” Tensor stores the step number of the last update of EV, `shape=[N]`, `dtype=int64`

You can read the value of checkpoint through the sdk of tensorflow to obtain the information corresponding to EV.

The first dimension shape of the above group of 4 tensors is the same, and they correspond in order. The EV of the partition is set, the number of tensors of the part depends on the set partitioner, and the sparse parameter of this feature is all tensor collections of all parts.



docs\_zh/Embedding-Variable/img\_2.jpg

Execute the above code to get the following results:



### 3.21.2 Example

#### Save Checkpoint

```
import tensorflow as tf

a = tf.get_embedding_variable('a', embedding_dim=4)
b = tf.get_embedding_variable('b', embedding_dim=8, partitioner=tf.fixed_size_
    ↪partitioner(4))

emb_a = tf.nn.embedding_lookup(a, tf.constant([0,1,2,3,4], dtype=tf.int64))
emb_b = tf.nn.embedding_lookup(b, tf.constant([5,6,7,8,9], dtype=tf.int64))

emb=tf.concat([emb_a, emb_b], axis=1)
loss=tf.reduce_sum(emb)

optimizer=tf.train.AdagradOptimizer(0.1)
train_op=optimizer.minimize(loss)

saver=tf.train.Saver()
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print(sess.run([emb, train_op]))
    saver.save(sess, "./ckpt_test/ckpt/model")
```

#### Restore Checkpoint

```
import tensorflow as tf
from tensorflow.core.protobuf.meta_graph_pb2 import MetaGraphDef
from tensorflow.python.framework import meta_graph

ckpt_path='ckpt_test/ckpt'
path=ckpt_path+'/model.meta'
meta_graph = meta_graph.read_meta_graph_file(path)
ev_node_list=[]
for node in meta_graph.graph_def.node:
    if node.op == 'KvVarHandleOp':
        ev_node_list.append(node.name)

print("ev node list", ev_node_list)
# filter ev-slot
non_slot_ev_list=[]
for node in ev_node_list:
    if "Adagrad" not in node:
        non_slot_ev_list.append(node)
print("ev (exculde slot) node list", non_slot_ev_list)

for name in non_slot_ev_list:
    print(name+'-keys', tf.train.load_variable(ckpt_path, name+'-keys'))
    print(name+'-values', tf.train.load_variable(ckpt_path, name+'-values'))
    print(name+'-freqs', tf.train.load_variable(ckpt_path, name+'-freqs'))
    print(name+'-versions', tf.train.load_variable(ckpt_path, name+'-versions'))
```

## Result

```

root@i22b15440:/home/admin/chen.ding/gitlab/code/DeepRec# python ckpt_test/gen_ckpt.py
2022-04-18 08:56:37.462789: I tensorflow/core/platform/profile_utils/cpu_utils.cc:94]
↳CPU Frequency: 2499445000 Hz
2022-04-18 08:56:37.468578: I tensorflow/compiler/xla/service/service.cc:168] XLA_
↳service 0x7fd5bb36fc60 initialized for platform Host (this does not guarantee that XLA_
↳will be used). Devices:
2022-04-18 08:56:37.469203: I tensorflow/compiler/xla/service/service.cc:176] _
↳StreamExecutor device (0): Host, Default Version
[array([[ 0.19091757, -1.2494173, -1.1098509, -0.88375354,  1.5314401,
         0.05683817, -0.3698739,  1.7533029, -0.95090204,  0.597882,
        -0.39382792,  1.2318059],
        [ 0.19091757, -1.2494173, -1.1098509, -0.88375354, -0.83169323,
         0.15894873, -0.66453475,  0.84301287,  1.125458,  0.12537971,
         0.7338474, -0.02672509],
        [ 0.19091757, -1.2494173, -1.1098509, -0.88375354, -0.29247162,
        -1.1461716,  1.1172409,  1.9220417, -1.2039331, -1.248681,
        -1.9431682, -0.27165115],
        [ 0.19091757, -1.2494173, -1.1098509, -0.88375354, -0.78701115,
        -0.28825614,  1.1483766, -0.18648145,  0.7928211, -0.4237969,
        -0.00831279,  0.68605185],
        [ 0.19091757, -1.2494173, -1.1098509, -0.88375354,  0.5981304,
         0.9115529,  1.2290154,  1.329322,  0.81167835,  0.43949136,
         0.4266789,  0.5895692]], dtype=float32), None]

root@i22b15440:/home/admin/chen.ding/gitlab/code/DeepRec# ls ckpt_test/ckpt/
checkpoint model.data-000000-of-000001 model.index model.meta

root@i22b15440:/home/admin/chen.ding/gitlab/code/DeepRec# python ckpt_test/read_ckpt.py
ev node list ['a', 'b/part_0', 'b/part_1', 'b/part_2', 'b/part_3', 'a/Adagrad', 'b/part_
↳0/Adagrad', 'b/part_1/Adagrad', 'b/part_2/Adagrad', 'b/part_3/Adagrad']
ev (exculde slot) node list ['a', 'b/part_0', 'b/part_1', 'b/part_2', 'b/part_3']
a-keys [0 1 2 3 4]
a-values [[ 0.19091757 -1.2494173 -1.1098509 -0.88375354]
 [ 0.19091757 -1.2494173 -1.1098509 -0.88375354]
 [ 0.19091757 -1.2494173 -1.1098509 -0.88375354]
 [ 0.19091757 -1.2494173 -1.1098509 -0.88375354]
 [ 0.19091757 -1.2494173 -1.1098509 -0.88375354]]
a-freqs []
a-versions []
b/part_0-keys [8]
b/part_0-values [[-0.8823574 -0.3836024  1.0530304 -0.28182772  0.69747484 -0.51914316
 -0.10365905  0.5907056]]
b/part_0-freqs []
b/part_0-versions []
b/part_1-keys [5 9]
b/part_1-values [[ 1.4360939 -0.03850809 -0.46522018  1.6579567 -1.0462483  0.5025357
 -0.4891742  1.1364597]
 [ 0.50278413  0.81620663  1.1336691  1.2339758  0.7163321  0.3441451
  0.33133262  0.49422294]]
b/part_1-freqs []
b/part_1-versions []

```

(continues on next page)

(continued from previous page)

```

b/part_2-keys [6]
b/part_2-values [[-0.83169323  0.15894873 -0.66453475  0.84301287  1.125458  0.12537971
  0.7338474 -0.02672509]]
b/part_2-freqs []
b/part_2-versions []
b/part_3-keys [7]
b/part_3-values [[-0.3878179 -1.2415178  1.0218947  1.8266954 -1.2992793 -1.3440272
 -2.0385144 -0.36699742]]
b/part_3-freqs []
b/part_3-versions []

```

## 3.22 AdamAsync Optimizer

### 3.22.1 Introduction

In the process of large-scale distributed asynchronous training, there are some problems in the implementation of Adam Optimizer of native Tensorflow, such as the speed of distributed training cannot be improved, and the load value of some PS nodes is very high.

To solve the problems encountered by Adam Optimizer during asynchronous training, we implemented AdamAsyncOptimizer:

1. Create associated beta1\_power and beta2\_power slots for each variable, thereby removing global dependencies;
2. When the Optimizer applies the gradient to a variable, it updates its associated beta1\_power and beta2\_power at the same time;
3. The calculation formula of adam is modified to the original version to solve the NAN problem;
4. The revised calculation formula is as follows:

```

auto alpha = lr() * Eigen::numext::sqrt(T(1) - beta2_power(0)) /
              (T(1) - beta1_power(0));

// beta1 ==  $\mu$ 
// beta2 ==  $\nu$ 
// v      == n
// var    ==  $\theta$ 
m.device(d) = m * beta1() + grad * (T(1) - beta1());
v.device(d) = v * beta2() + grad.square() * (T(1) - beta2());
if (use_nesterov) {
    var.device(d) -= ((grad * (T(1) - beta1()) + beta1() * m) * alpha) /
                    (v.sqrt() + epsilon());
} else {
    var.device(d) -= (m * alpha) / (v.sqrt() + epsilon());
}

// update beta1_power && beta2_power
beta1_power.device(d) = beta1_power * beta1();
beta2_power.device(d) = beta2_power * beta2();

```

5. For sparse variables, when applying gradient, doing momentum will reduce the update rate of sparse features;

6. When applying sparse variables, we provide a bool variable (default false). When set to true, the update algorithm can be changed from adam to rmsprop, so that the sliding average function of momentum can be removed.
7. AdamAsync Optimizer is used in the same way as AdamOptimizer, and there is an additional configurable parameter: `apply_sparse_rmsprop`, whether to enable the rmsprop algorithm when apply sparse is disabled by default.

### 3.22.2 User interface

You need to use the `tf.train.AdamAsyncOptimizer` interface during training, which is the same as other TF native Optimizers. The specific definition is as follows:

```
class AdamAsyncOptimizer(optimizer.Optimizer):
def __init__(self, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8,
              use_locking=False, apply_sparse_rmsprop=False, name="AdamAsync"):

# call function
optimizer = tf.train.AdamAsyncOptimizer(
    learning_rate_new,
    beta1=0.9,
    beta2=0.999,
    epsilon=1e-8)
```

### 3.22.3 Example

```
import tensorflow as tf

var = tf.get_variable("var_0", shape=[10,16],
                      initializer=tf.ones_initializer(tf.float32))

emb = tf.nn.embedding_lookup(var, tf.cast([0,1,2,5,6,7], tf.int64))
fun = tf.multiply(emb, 2.0, name='multiply')
loss = tf.reduce_sum(fun, name='reduce_sum')

gs= tf.train.get_or_create_global_step()
opt = tf.train.AdamAsyncOptimizer(0.1)

g_v = opt.compute_gradients(loss)
train_op = opt.apply_gradients(g_v)

init = tf.global_variables_initializer()

sess_config = tf.ConfigProto(allow_soft_placement=True, log_device_placement=False)
with tf.Session(config=sess_config) as sess:
    sess.run([init])
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
```

## 3.23 AdagradDecay Optimizer

### 3.23.1 Introduction

AdagradDecay optimizer is an improved version of the Adagrad optimizer proposed to support ultra-large-scale training and has a good effect in recommendation and search scenarios.

In addition to its complexity, the ultra-large-scale model generally has the following characteristics:

1. model training needs a huge amount of samples, each training needs more than 1 billion samples.
2. The continuous incremental training of the model takes a long time, up to more than one month. (Practice has proved that the accumulation of data can improve the effectiveness of the model).

The current Adagrad optimizer cannot cope with such huge data.

### 3.23.2 The principle of Adagrad

$$\begin{aligned} v_t &= v_{t-1} + g_t^2 \\ \Delta w &= \frac{\eta}{\sqrt{v_t + \epsilon}} g_t \end{aligned}$$

With the accumulation of data,  $v_t$  will tend to infinity, causing  $\Delta w$  to approach 0, which means that new data cannot affect the model.

Some studies have improved the update method of  $v_t$  as follows:

$$v_t = \rho v_{t-1} + (1 - \rho) g_t^2$$

That is, the cumulant is discounted at each iteration. This will solve the problem of  $v_t$  going to infinity, but it will also cause the model to perform poorly. The reason is the inappropriate use of samples.

Assuming 1 billion samples per day, the training batch size is 1000, so one day's data needs 1000,000 iterations to complete the training. Assuming  $\rho = 0.9999$ , the sample of the first batch has an effect of  $0.9999^{1000000} = 3.7015207857933866e - 44$ , closes to 0, which is unreasonable.

The effect of the sample should be learned by the model, rather than relying on the artificial order and rules. Moreover, considering the objective environment in which the samples were produced, there is not such a large difference between the first sample and the subsequent samples, so the strategy of discounting by iteration is not appropriate.

We propose the concept of discounting according to the sampling period. The sample discount in the same period is

the same, taking into account the infinite accumulation of data and the impact of sample order on the model.  $\rho(t)$  is defined as follows:

$$\rho(t) = \begin{cases} \rho, & t \bmod T = 0 \\ 1.0, & \text{otherwise} \end{cases}$$

so,  $v_t = \rho(t)v_{t-1} + g_t^2$ .

Where T is the discount period. In practice, it can be set according to the pattern of sample generation and the impact on the model. For example, the daily purchase behavior of users can be roughly divided into several periods: early morning, morning, afternoon, and evening. The discount period can be set to 1/4 of each day. When all samples are

trained, the part in the early morning is only discounted by  $\rho^4$ .

T handles sparse features in the same way as dense features, it is global T, not the number of times sparse features appear. The purpose of this is to speed up the model's learning of sparse features.

The discount will not be unlimited. In order to avoid the problem that  $v_t$  is too small due to the discount, we have introduced protection measures, that is,  $v_t$  has a lower limit protection.

### 3.23.3 User API

Users only need to define `tf.train.AdagradDecayOptimizer` during training, which is the same as other TF native optimizers.

```
class AdagradDecayOptimizer(optimizer.Optimizer):
    """Optimizer that implements the Adagrad algorithm with accumulator decay.
    Different from the original Adagrad algorithm, AdagradDecay performs decay
    at given step with given rate. So that the accumulator will not be infinity.
    """

    def __init__(self,
                 learning_rate,
                 global_step,
                 initial_accumulator_value=0.1,
                 accumulator_decay_step=1000000,
                 accumulator_decay_rate=0.9,
                 use_locking=False,
                 name="AdagradDecay"):
        """Construct a new AdagradDecay optimizer.

    Args:
        learning_rate: A `Tensor` or a floating point value. The learning rate.
        global_step: global step variable, used for calculating t%T .
        initial_accumulator_value: A floating point value. Starting and baseline
            value for the accumulators, must be positive. The accumulators will not
            be less than it.
        accumulator_decay_step: When global_step reaches times of
            accumulator_decay_step, accumulator will be decayed with
            accumulator_decay_rate. accumulator *= accumulator_decay_rate
```

(continues on next page)

(continued from previous page)

```

accumulator_decay_rate: Decay rate as above described.
use_locking: If `True` use locks for update operations.
name: Optional name prefix for the operations created when applying
      gradients. Defaults to "AdagradDecay".

Raises:
  ValueError: If the `initial_accumulator_value`, `accumulator_decay_step`
              or `accumulator_decay_rate` is invalid.
"""

```

### 3.23.4 Example

```

import tensorflow as tf

var = tf.get_variable("var_0", shape=[10,16],
                      initializer=tf.ones_initializer(tf.float32))

emb = tf.nn.embedding_lookup(var, tf.cast([0,1,2,5,6,7], tf.int64))
fun = tf.multiply(emb, 2.0, name='multiply')
loss = tf.reduce_sum(fun, name='reduce_sum')

gs= tf.train.get_or_create_global_step()
opt = tf.train.AdagradDecayOptimizer(0.1, global_step=gs)

g_v = opt.compute_gradients(loss)
train_op = opt.apply_gradients(g_v)

init = tf.global_variables_initializer()

sess_config = tf.ConfigProto(allow_soft_placement=True, log_device_placement=False)
with tf.Session(config=sess_config) as sess:
    sess.run([init])
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))

```

## 3.24 AdamW Optimizer

### 3.24.1 Introduction

The AdamW optimizer supports EmbeddingVariable, which adds the weight decay function compared to the Adam optimizer.

This is a kind of implementation of the AdamW optimizer which is mentioned in Loshchilov & Hutter (<https://arxiv.org/abs/1711.05101>) "Decoupled Weight Decay Regularization".

### 3.24.2 User interface

You need to use the `tf.train.AdamWOptimizer` function interface during training, which is the same as other TF native Optimizers. The specific definition is as follows:

```
class AdamWOptimizer(DecoupledWeightDecayExtension, adam.AdamOptimizer):
    def __init__(self,
                  weight_decay,
                  learning_rate=0.001,
                  beta1=0.9,
                  beta2=0.999,
                  epsilon=1e-8,
                  use_locking=False,
                  name="AdamW"):

# call function
optimizer = tf.train.AdamWOptimizer(
    weight_decay=weight_decay_new
    learning_rate=learning_rate_new,
    beta1=0.9,
    beta2=0.999,
    epsilon=1e-8)
```

### 3.24.3 Example

```
import tensorflow as tf

var = tf.get_variable("var_0", shape=[10,16],
                      initializer=tf.ones_initializer(tf.float32))

emb = tf.nn.embedding_lookup(var, tf.cast([0,1,2,5,6,7], tf.int64))
fun = tf.multiply(emb, 2.0, name='multiply')
loss = tf.reduce_sum(fun, name='reduce_sum')

gs= tf.train.get_or_create_global_step()
opt = tf.train.AdamWOptimizer(weight_decay=0.01, learning_rate=0.1)

g_v = opt.compute_gradients(loss)
train_op = opt.apply_gradients(g_v)

init = tf.global_variables_initializer()

sess_config = tf.ConfigProto(allow_soft_placement=True, log_device_placement=False)
with tf.Session(config=sess_config) as sess:
    sess.run([init])
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
    print(sess.run([emb, train_op, loss]))
```



## 3.25 oneDNN

### 3.25.1 Introduction

oneDNN is the open source cross-platform performance acceleration library for deep learning from Intel, The [documentation](#) guides you to find out which primitives are supported. OneDNN has been integrated into DeepRec, which can be enabled by adding the compiling option in the compile command. `--config=mkl_threadpool` is used to enable oneDNN accelerated arithmetic computation. Adding the compiling option `--config=opt` will enable the optimization of `--copt=-march=native`, which can further accelerate arithmetic performance on the CPU which supports AVX512, for example, Skylake, Caslake and Icelake.

Tips: MKL was first renamed as DNNL and then renamed as oneDNN. Tensorflow initially used MKL to accelerate the computation of the operators, and in subsequent versions of iteration, oneDNN gradually take the place of MKL, but the macro definitions were still retained.

Macro definition of oneDNN in DeepRec:

Macro Definition	Values (Bold for Default)	Explanation
TF_MKL_PRIMITIVE_REPLACE_FOR_RECO	1, 0, <b>0</b>	1: Only replace the operators which supported by oneDNN in recommendation models; 0: Replace all of the operators to that supported by oneDNN.
TF_MKL_OPTIMIZE_PRIMITIVE_MEMUSE	1, 0, <b>0</b>	1: Reduce the use of main memory by releasing the primitives; 0: Don't release primitives.
TF_DISABLE_MKL	<b>0</b> , 1	0: Enable MKL; 1: Disable MKL
TF_MKL_NUM_INTRA_THREADS	Integer, such as 14, <b>Not set by default</b>	Integer: set the number of intra threads used by oneDNN; Not set: number of TF intra threads used most.
ONEDNN_VERBOSE	0/1/2	Print the level of log output by oneDNN primitive.
DNNL_MAX_CPU_ISA	<b>ALL</b> , AVX512_CORE_AMX, AVX512_CORE_BF16, ...	The highest ISA used by oneDNN (for versions less than 2.5.0)
ONEDNN_MAX_CPU_ISA	<b>ISA</b> , AVX512_CORE_AMX, AVX512_CORE_BF16, ...	The highest ISA by oneDNN (for versions more than or equal to 2.5.0)

Primitives supported by oneDNN:

Primitive	Available Types	Available Backward Operations
Matrix Multiplication	f32, bf16, f16, u8, s8	Scale, Zero, Eltwise, Sum, Binary
Inner Product	f32, bf16, f16, u8, s8	Scale, Eltwise, Sum, Binary
Layer Normalization	f32, bf16, f16	/
Batch Normalization	f32, bf16, f16, s8	Eltwise
Local Response Normalization (LRN)	f32, bf16, f16	/
Binary (+, =, *, /, >, <, min, max...)	f32, bf16, f16, u8, s8	Scale, Eltwise, Sum, Binary
Eltwise (relu, gelu, tanh, linear...)	f32, s32, bf16, f16, u8, s8	Binary
PReLU	f32, s32, bf16, s8, u8	/
Sum	f32, s32, bf16, f16, u8, s8	/
Reduction	f32, bf16, u8, s8	Eltwise, Sum, Binary
Softmax	f32, bf16, f16	/
LogSoftmax	f32, bf16	/
Reorder	f32, s32, bf16, f16, u8, s8	Scale, Sum
Concat	f32, s32, bf16, f16, u8, s8	/
Convolution	f32, bf16, f16, u8, s8	Scale, Zero, Eltwise, Sum, Binary
Pooling	f32, s32, bf16, f16, u8, s8	Binary
RNN (LSTM, GRU, Vanilla RNN...)	f32, bf16, f16, u8, s8	/
Resampling	f32, s32, bf16, f16, s8, u8	Eltwise, Sum, Binary
Shuffle	f32, s32, bf16, s8, u8	/

## 3.26 Optimization of Operator

### 3.26.1 Hardware and Software Configuration

Hardware: Alibaba Cloud ECS general purpose instance family with high clock speeds - [ecs.hfg7.2xlarge](#).

CPU number: 8 cores

Baseline version: Tensorflow v1.15.5

Optimized version: DeepRec

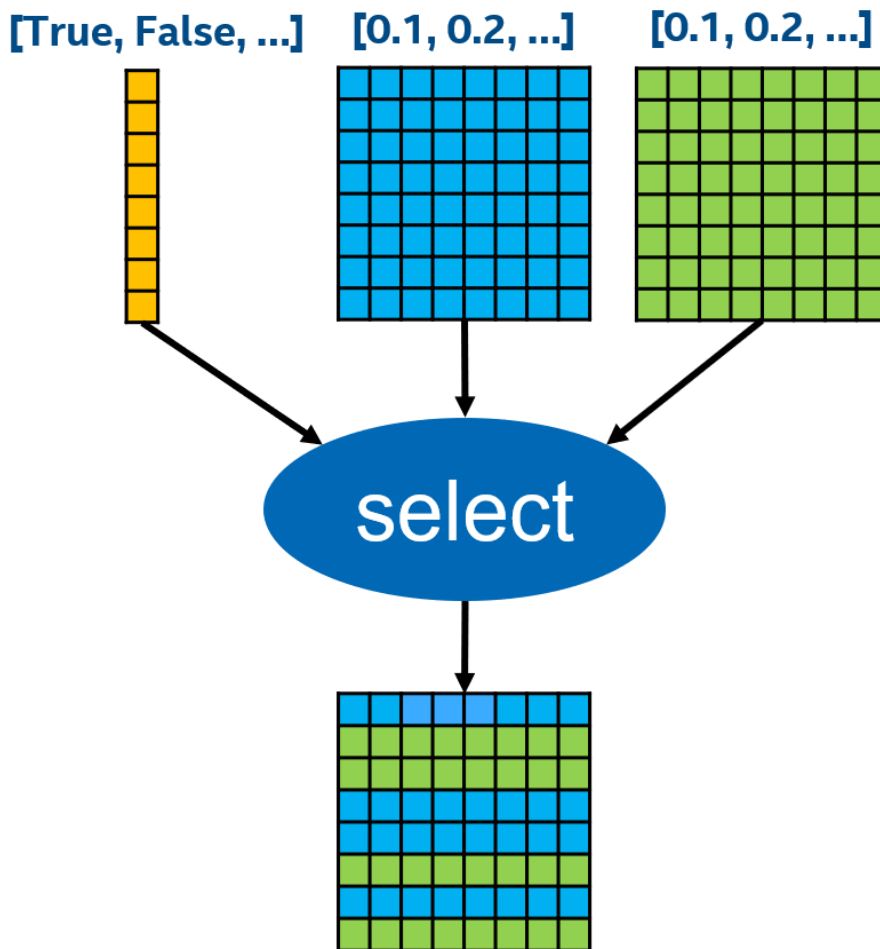
Gcc version 7.5.0

### 3.26.2 Performance Data

Op Name	Input Tensor Shape	Baseline Perf (latency/ms)	Optimized Perf (latency/ms)	Speedup
Select	condition: (1024, 64), x: (1024, 64), y: (1024, 64)	2.080	0.564	+3.68X
Dynamic_stitch	indices: (40, 2500), data: (40, 2500, 64)	82.14	24.77	+3.31X
Transpose	data: (1024, 64)	1.504	0.366	+4.11X
Tile	input: (512, 50), multiples: (2, 50)	1.68	0.125	+13.44X
BiasAddGrad	data: (51200, 512)	26.84	1.67	+16.07X
SparseSegmentMean	data: (51200, 128), indices: (51200), seg index: (51200)	1.93	0.445	+4.34X
Unique				
Gather				
BiasAdd				
where				
DynamicPartition				
SparseConcat				

### 3.26.3 Case Studies: Select

The computing process of operator Select:



TensorFlow original implementation: Broadcast + Elementwise Select

```
template <typename Device, typename T, int NDIMS>
struct BCastSelectFunctorBase {
    void operator()(const Device& d,
                    typename TTypes<T, NDIMS>::Tensor output_tensor,
                    typename TTypes<bool, NDIMS>::ConstTensor cond_tensor,
                    typename TTypes<T, NDIMS>::ConstTensor then_tensor,
                    typename TTypes<T, NDIMS>::ConstTensor else_tensor,
                    typename Eigen::array<Eigen::DenseIndex, NDIMS> cond_bcast,
                    typename Eigen::array<Eigen::DenseIndex, NDIMS> then_bcast,
                    typename Eigen::array<Eigen::DenseIndex, NDIMS> else_bcast) {
        output_tensor.device(d) = cond_tensor.broadcast(cond_bcast)
            .select(then_tensor.broadcast(then_bcast),
                   else_tensor.broadcast(else_bcast));
    }
};
```

PAI-TF (Merged to Community): Row Select ,Optimized redundant broadcast operations in the original TensorFlow version. ◦

```

if (c[i]) {
    for (size_t j = 0; j < batch_size; ++j) {
        output[offset + j] = t[offset + j];
    }
} else {
    for (size_t j = 0; j < batch_size; ++j) {
        output[offset + j] = e[offset + j];
    }
}

```

DeepRec: vectorized Row Select, used AVX512 mask vectorisation instructions for the further optimizing of select operation, which improved the performance of this operator by 3.68x.

```

__mmask16 cmask = (c[i] == false) ? 0xffff : 0x0000; // select t/e
size_t ofs = 0;

for (size_t j = 0; j < quotient; ++j) {
    __m512 src = _mm512_loadu_ps(t + offset + ofs);
    __m512 tmp = _mm512_mask_loadu_ps(src, cmask, e + offset + ofs);
    _mm512_storeu_ps(output + offset + ofs, tmp);
    ofs += float_alignment;
}

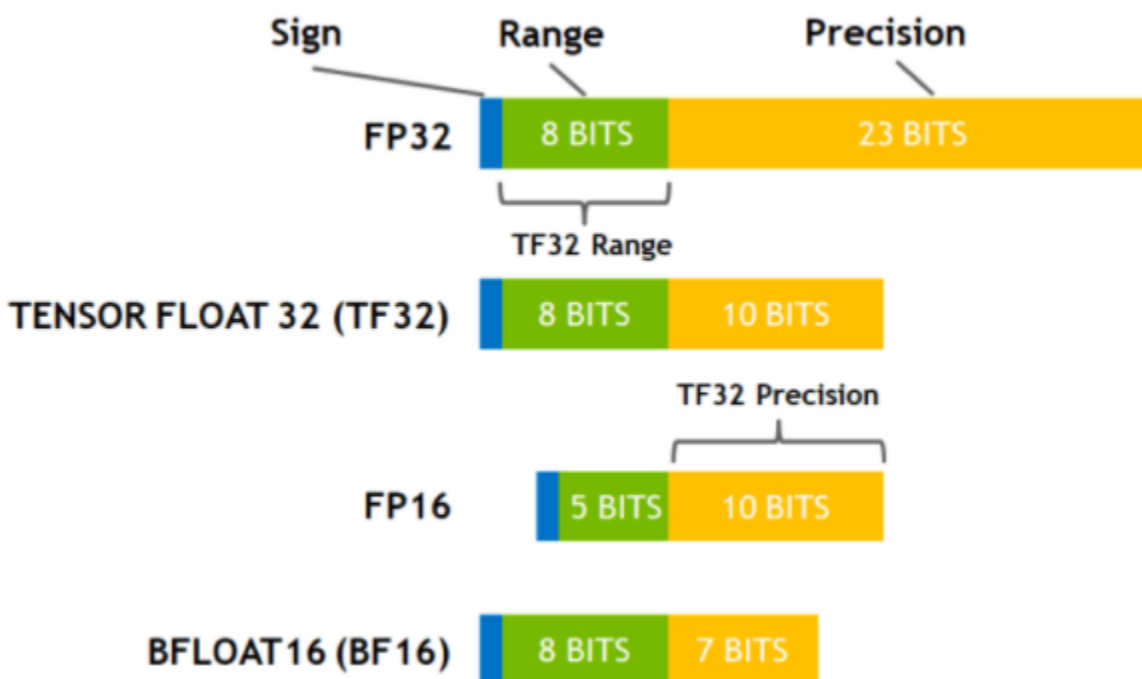
if (remainder != 0) {
    __mmask16 mask = (remainder >= float_alignment)
        ? 0xffff : 0xffff >> (float_alignment - remainder);
    cmask &= mask;
    __m512 src = _mm512_mask_loadu_ps(_mm512_setzero_ps(), mask, t + offset + ofs);
    __m512 tmp = _mm512_mask_loadu_ps(src, cmask, e + offset + ofs);
    _mm512_mask_storeu_ps(output + offset + ofs, mask, tmp);
}

```

## 3.27 NVIDIA TF32

### 3.27.1 TF32 Intorduction

TensorFloat-32 is a computational format that is specifically designed for use with TensorCore on Nvidia's Ampere architecture GPUs. When comparing with other common data formats:



On A100, using TF32 for matrix multiplication can provide 8x faster performance than using FP32 CUDA Core on V100. Note that TF32 is only a intermediate computation format when using TensorCore.

### 3.27.2 Usage and Prerequisites

Prerequisites:

1. Matrix multiplication and convolution-related operations, and input data type is FP32, can use TF32 as TensorCore's intermediate computation type.
2. Ampere architecture GPU

Typically, the use of TF32 is used automatically by cuBlas, cuDNN, and other Nvidia computation libraries internally. Therefore, regardless of the deep learning framework you use on top, make sure:

1. cuBLAS  $\geq$  11.0
2. cuDNN  $\geq$  8.0

On Ampere architecture GPU, TF32 is enabled by default for computation acceleration. However, not every matrix and convolution computation will necessarily use TF32, depending on factors such as the input data type and shape. TF32 will be used as much as possible when appropriate.

To forcefully disable TF32, you can set the environment variable:

```
export NVIDIA_TF32_OVERRIDE=0
```

to command all Nvidia libraries to turn off TF32.

### 3.27.3 Impact of TF32 on precision

Nvidia has compared the precision difference of various well-known models between FP32 and TF32 and found that TF32 has little impact on precision:

Classification Tasks				Detection & Segmentation Tasks					Language Tasks					
Architecture	Network	Top-1 Accuracy		Architecture	Network	Metric	Model Accuracy		Architecture	Network	Dataset	Metric	Model Accuracy	
		FP32	TF32				FP32	TF32					FP32	TF32
ResNet	RN18	70.43	70.58	Faster RCNN	RN50 FPN 1X	mAP	37.81	37.95	Transformer	Vaswani Base	WMT	BLEU	27.18	27.10
	RN32	74.03	74.08		RN101 FPN 3X	mAP	40.04	40.19		Vaswani Large	WMT	BLEU	28.63	28.62
	RN50	76.78	76.73		RN50 FPN 3X	mAP	42.05	42.14		Levenshtein	WMT	Loss	6.16	6.16
	RN101	77.57	77.57		TorchVision	mAP	37.89	37.89		Light Conv Base	WMT	BLEU	28.55	28.74
ResNext	RNX50	77.51	77.62	Mask RCNN		mIOU	34.65	34.69	Convolutional	Light Conv Large	WMT	BLEU	30.10	30.20
	RNX101	79.10	79.30		RN50 FPN 1X	mAP	38.45	38.63		Dynamic Conv Base	WMT	BLEU	28.34	28.42
WideResNet	WRN50	77.99	78.11			mIOU	35.16	35.25		Dynamic Conv Large	WMT	BLEU	30.10	30.31
	WRN101	78.61	78.62		RN50 FPN 3X	mAP	41.04	40.93		FairSeq Conv	WMT	BLEU	24.83	24.86
DenseNet	DN121	75.57	75.57			mIOU	37.15	37.23	Recurrent	GNMT	WMT	BLEU	24.53	24.80
	DN169	76.75	76.69		RN101 FPN 3X	mAP	42.99	43.08	Convolutional Transformer	Fairseq Dauphin	WikiText	PPL	35.89	35.80
VGG	V11-BN	68.47	68.44			mIOU	38.72	38.73		XL Standard	WikiText	PPL	22.89	22.80
	V16-BN	71.54	71.51	Retina Net	RN50 FPN 1X	mAP	36.46	36.49	BERT	Base Pre-train	Wikipedia	LM Loss	1.34	1.34
	V19-BN	72.54	72.68		RN50 FPN 3X	mAP	38.04	38.19		Base	SQUAD v1 F1		87.95	87.66
	V19	71.75	71.60		RN101 FPN 3X	mAP	39.75	39.82		Downstream	SQUAD v2 F1		76.68	75.67
GoogLeNet	InceptionV3	77.20	77.34	RPN	RN50 FPN 1X	mAP	58.02	58.11						
	Xception	79.09	79.31	Single-Shot Detector (SSD)	RN18	mAP	19.13	19.18						
Dilated RN	DRN A 50	78.24	78.16		RN50	mAP	24.91	24.85						
ShuffleNet	V2-X1	68.62	68.87											
	V2-X2	73.02	72.88											
MNASNet	V1.0	71.62	71.49											
SqueezeNet	V1_1	60.90	60.85											
MobileNet	MN-V2	71.64	71.76											
Stacked UNet	SUN64	69.53	69.62											
EfficientNet	B0	76.79	76.72											

### 3.27.4 More Info

1. <https://developer.nvidia.com/blog/accelerating-ai-training-with-tf32-tensor-cores/>
2. <https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/>

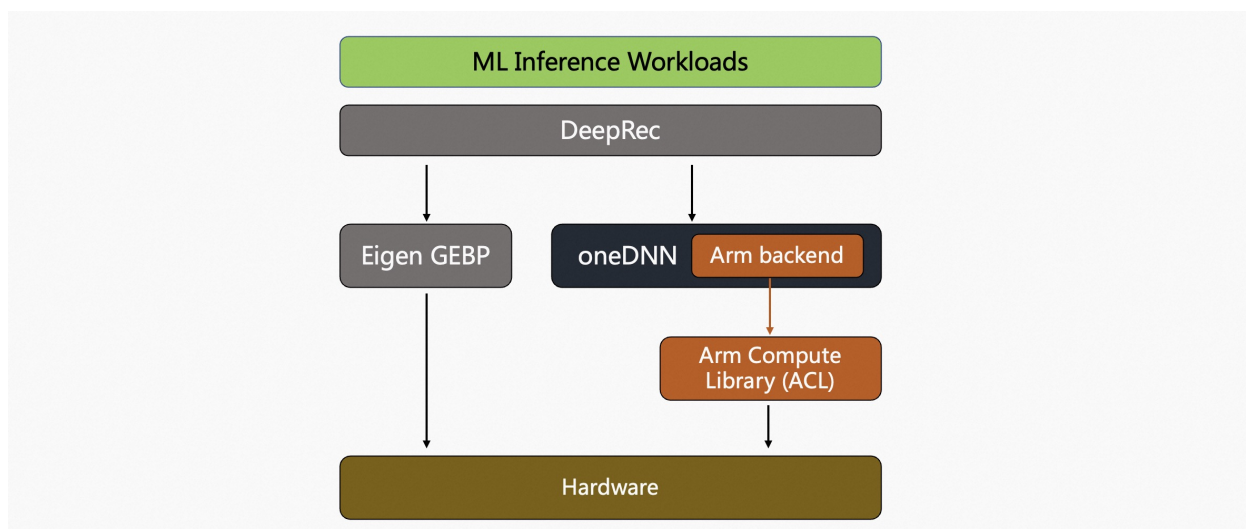
## 3.28 Arm Compute Library

### 3.28.1 Introduction

The **Arm Compute Library** (ACL) is an open-source high-performance computing library developed by ARM, aimed at optimizing compute-intensive tasks on processors based on the ARM architecture. It provides a range of compute functions and algorithms, including implementations of many modern machine learning algorithms, which can be used for tasks in areas such as deep learning. The [documentation](#) provides information on the compute primitives supported by ACL.

Currently, ACL support has been added to DeepRec, and ACL acceleration operators can be enabled simply by adding the ACL compilation option `--config=mk1_aarch64` to the DeepRec compilation command.

The integration of ACL into DeepRec is achieved through oneDNN as an intermediate calling layer, as shown in the fol-



lowing figure:

The behavior of ACL can also be controlled through oneDNN macros. For example, users can enable low-precision inference acceleration on devices that support bf16:

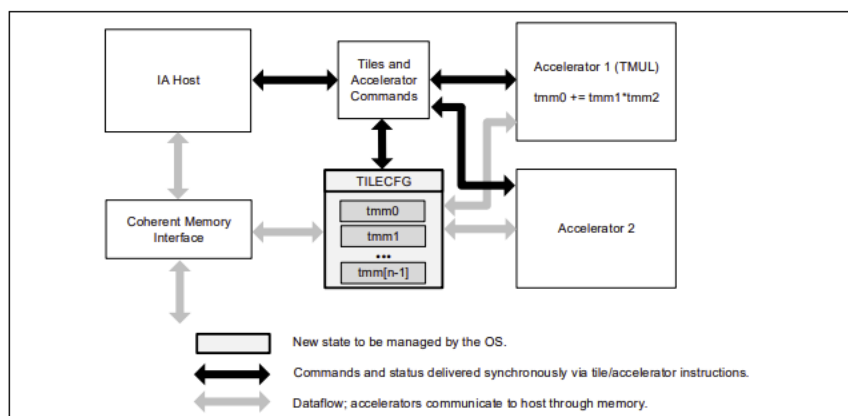
```
ONEDNN_DEFAULT_FPMATH_MODE=BF16 python foo.py
```

By specifying the environment variable `ONEDNN_DEFAULT_FPMATH_MODE=BF16`, ACL will automatically determine whether the hardware device supports BF16 matrix multiplication instructions. If so, FP32 matrix multiplication will be replaced with BF16 implementation.

### 3.29 Intel® AMX

Intel® Advanced Matrix Extensions (Intel® AMX) is the new accelerating technology for Deep Learning, which is supported on the fourth-generation Intel® Xeon® Scalable processor Sapphire Rapids [AliCloud g8i specification family](#). It supports two datatypes, BFloat16 and Int8. The details about BFloat16 usage, please refer to [Bfloat16](#).

Intel® Advanced Matrix Extensions (Intel® AMX) is a new 64-bit programming paradigm consisting of two components: a set of 2-dimensional registers (tiles) representing sub-arrays from a larger 2-dimensional memory image and an accelerator able to operate on tiles. The first implementation is called TMUL (tile matrix multiply unit). Below shows the AMX architecture.





### 3.29.1 Requirements and Methods

#### Requirements:

The cloud instance requires to be the fourth-generation Intel® Xeon® Scalable processor Sapphire Rapids [AliCloud g8i specification family](#). It also requires to use DeepRec which is compiled and optimized by oneDNN in order to provide AMX instruction acceleration, details of which can be found in the oneDNN section. Except for that, there are other software requirements:

- OS or Kernel version: The kernel should be 5.16+. Both of Anolis OS and Ubuntu2204 support AMX.
- Python Version: If python is used, the version should be python3.8+. If C++ is used, then it's fine.

#### Methods:

As the recommendation scenarios are extremely demanding in terms of model accuracy, in order to improve model performance while taking into account model accuracy, AMX BFloat16 can be used in recommendation system. There are two methods to make use of AMX.

#### Enable AMX by the special BFloat16 API (Only for training)

The details are described in [Bfloat16](#) section. The advantage is that, developers can decide which graph will run in BF16 and which graph will keep FP32 in order to meet the accuracy requirement. And the introduced cast overhead can be minimized by operator fusion to achieve the better performance. But if users want to get benefit from AMX more easier, they can try the second method as below.

#### Enable AMX by setting the runtime environment ONEDNN\_DEFAULT\_FPMATH\_MODE=bf16(For both training and inference)

The method is supported by the oneDNN library without requiring any code modification from the user. It automatically recognizes matrix multiplication and accelerates computation using the AMX BF16 instruction set. It is very convenient to use, but the performance and precision may be slightly inferior to the first method. The specific effects depend on the actual model.

```
export ONEDNN_DEFAULT_FPMATH_MODE=bf16
```

Besides, users can confirm whether AMX is working by setting the ONEDNN\_VERBOSE environment variable. If it is working, keywords such as 'AMX' will be printed as output.

```
export ONEDNN_VERBOSE=1
```

### 3.29.2 Performance Comparison

The following provides the AMX performance data comparison by using BFloat16 API to freely control the computational graph.

Use models in DeepRec Modelzoo to compare the DeepRec with BF16 and FP32 to see the performance improvement. Models in Modelzoo can enable the BF16 feature by adding --bf16 parameter.

Use Aliyun ECS cloud server as benchmark machine, Intel 4th Xeon Scalable Processor(Sapphire Rapids) specification [AliCloud g8i specification family](#)

- Hardware configuration:

- Intel(R) Xeon(R) Platinum 8475B
- CPU(s): 16
- Socket(s): 1
- Core(s) per socket: 8
- Thread(s) per core: 2
- Memory: 64G
- Software configuration:
  - kernel: Linux version 5.15.0-58-generic
  - OS: Ubuntu 22.04.2 LTS
  - GCC: 11.3.0
  - Docker: 20.10.21
  - Python: 3.8

Performance Result:

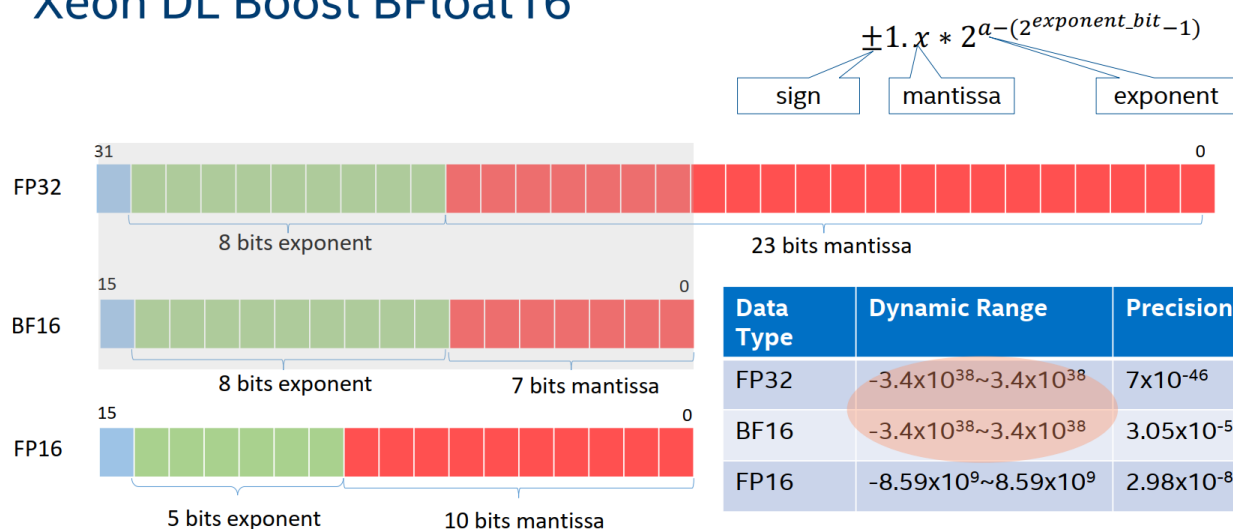
Throughput	WDL	MMoE	DSSM
FP32	33605.22	69538.47	102072.3
FP32+BF16	85801.28	155582	209099.7
Speedup	2.55x	2.24x	2.05x

BF16 has little effect on the AUC metric of model training, more details of the difference can be found in the documentation of each model in the model zoo.

## 3.30 BFloat16

BFloat16 (BF16) is a computational format and the instruction for accelerating deep learning training and inference, which is supported on the third-generation Intel® Xeon® Scalable processor Cooper Lake [AliCloud hfg7 specification family](#) and its successor processors. Below shows the comparison with other commonly used data formats:

## Xeon DL Boost BFloat16



BF16 favors Dynamic Range than Precision.

### 3.30.1 Requirments and methods

**Requirements:** The cloud instance requires to be the third-generation Intel® Xeon® Scalable processor Cooper Lake [AliCloud hfg7 specification family](#). It also requires to use DeepRec which is compiled and optimized by oneDNN in order to provide BF16 instruction acceleration, details of which can be found in the oneDNN section.

**Method:** As the recommended scenarios are extremely demanding in terms of model accuracy, in order to improve model performance while taking into account model accuracy, users could control the BF16 computing graph freely in the following way:

- Step 1: Add `.keep_weights(dtype=tf.float32)` after `tf.variable_scope(...)` to keep the current weights as FP32.
- Step 2: Add `tf.cast(..., dtype=tf.bfloat16)` to transfer the input tensors to BF16 type.
- Step 3: Add `tf.cast(..., dtype=tf.float32)` to transfer the output tensors to FP32 type.

```
with tf.variable_scope(...).keep_weights(dtype=tf.float32):
    inputs_bf16 = tf.cast(inputs, dtype=tf.bfloat16)
    ... // BF16 graph, FP32 weights
    outputs = tf.cast(outputs_bf16, dtype=tf.float32)
```

Example:

```
import tensorflow as tf

inputs = tf.ones([4, 8], tf.float32)

with tf.variable_scope('dnn', reuse=tf.AUTO_REUSE).keep_weights(dtype=tf.float32):
    # cast inputs to BF16
    inputs = tf.cast(inputs, dtype=tf.bfloat16)
    outputs = tf.layers.dense(inputs, units=8, activation=tf.nn.relu)
    outputs = tf.layers.dense(inputs, units=1, activation=None)
    # cast ouputs to FP32
```

(continues on next page)

(continued from previous page)

```
outputs = tf.cast(outputs, dtype=tf.float32)

outputs = tf.nn.softmax(outputs)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print(sess.run(outputs))
```

Special Reminder: according to the experience of parameters tuning, usually the last layer of DNN network in a multi-layer DNN network has the most impact on the accuracy of the model, which occupies a lower computational ratio. So the last layer of DNN network can be converted to FP32 type to run, which can improve the computational performance of the model training while preserving the accuracy of the model.

To maintain consistency with the accuracy of the model without BF16 optimization, DeepRec provides the `keep_weights(dtype=dtypes.float32)` method in `variable_scope`. With this method, all variables in this variable field will be saved in FP32 format, which significantly reduce the cumulative summation error of variables. And the cast operation is automatically added to the graph, converting it to BF16 format for computation. To reduce the extra computational overhead of the cast operation introduced, DeepRec automatically fuses the cast operator with its nearest operator to improve the operation performance. DeepRec will perform the following fusion operations on cast-related operators.

- MatMul + Cast
- Concat + Cast
- Split + Cast

### 3.30.2 Performance comparison

Use models in DeepRec Modelzoo to compare the DeepRec with BF16 and FP32 to see the performance improvement. Models in Modelzoo can enable the BF16 feature by adding `--bf16` parameter.

Use Aliyun ECS cloud server as benchmark machine, Intel 3rd Xeon Scalable Processor(Cooper Lake) specification [ecs.hfg7.2xlarge](#)

- Hardware configuration:
  - Intel(R) Xeon(R) Platinum 8369HC CPU @ 3.30GHz
  - CPU(s): 8
  - Socket(s): 1
  - Core(s) per socket: 4
  - Thread(s) per core: 2
  - Memory: 32G
- Software configuration:
  - kernel: 4.18.0-348.2.1.el8\_5.x86\_64
  - OS: CentOS Linux release 8.5.2111
  - GCC: 8.5.0
  - Docker: 20.10.12
  - Python: 3.6.8

Performance Result:

Throughput	WDL	DeepFM	DSSM
FP32	15792.49	30718.6	114436.87
FP32+BF16	22633.8	34554.67	125995.83
Speedup	1.43x	1.12x	1.10x

BF16 has little effect on the AUC metric of model training, more details of the difference can be found in the documentation of each model in the model zoo.

## 3.31 WorkQueue

### 3.31.1 Introduction

In large-scale distributed asynchronous training, if different workers read the same number of samples, the training time of the slow node will be much longer than that of other nodes, resulting in a long-tail problem. With the expansion of the training scale, the long-tail problem will become more and more serious, which reducing the overall data throughput, and prolonging the time to produce the model.

We provide the WorkQueue class, which can perform elastic data segmentation on multiple data sources, so that slow nodes can be trained with less data, and fast nodes can be trained with more data. WorkQueue will significantly alleviate the impact of long-tail problems and reduce training time.

WorkQueue manages the work items of all workers. After the remaining work items are consumed, each worker will obtain new work items from the same WorkQueue as a data source for training, so that faster training workers can get more work items.

### User APIs

#### WorkQueue class definition

```
class WorkQueue(works, num_epochs=1,
                shuffle=True,
                seed=None,
                prefix=None,
                num_slices=None,
                name='work_queue')
```

- **works**: list of filename
- **num\_epochs**: the number of times to read all data
- **shuffle**: if True, randomly shuffle data every epoch
- **seed**: the random seed used to shuffle the data, If None, the seed will be automatically generated by WorkQueue
- **prefix**: the prefix of work items (filenames or table names), default value is None
- **num\_slices**: total number of work items, usually more than 10 times the number of workers. The more unstable the cluster, the greater the total number of work items required. If None, no data fragmentation will be performed. num\_slices is invalid when reading files.
- **name**: the name of work queue

## method introduction

- **take**

method *WorkQueue.take()*

Description	Get a work item from global WorkQueue and download it to the worker
Return Value	tensorflow.Tensor
Parameter	None

- **input\_dataset**

method *WorkQueue.input\_dataset()*

Description	Return a Dataset, Each element of the Dataset is a work item
Return Value	tensorflow.data.Dataset
Parameter	None

- **input\_producer**

method *WorkQueue.input\_producer()*

Description	The local proxy queue of the global work queue, used by the Reader class Op.
Return Value	tensorflow.FIFOQueue
Parameter	None

- **add\_summary**

method *WorkQueue.add\_summary()*

Description	Generates work queue statistics that can be displayed in tensorboard.
Return Value	None
Parameter	None

## 3.31.2 Examples

### Use tf.dataset as data sources

```
from tensorflow.python.ops.work_queue import WorkQueue

# use WorkQueue to allocate tasks
work_queue = WorkQueue([filename, filename1, filename2, filename3])
f_data = work_queue.input_dataset()
# Extract lines from input files using the Dataset API.
dataset = tf.data.TextLineDataset(f_data)
dataset = dataset.shuffle(buffer_size=20000,
                          seed=2021) # fix seed for reproducing
dataset = dataset.repeat(num_epochs)
dataset = dataset.prefetch(batch_size)
dataset = dataset.batch(batch_size)
dataset = dataset.prefetch(1)
```

The [WDL](#) model in DeepRec modelzoo provides a more detailed example.

### Use files as data sources

```
from tensorflow.python.ops.work_queue import WorkQueue

work_queue = WorkQueue([path1, path2, path3], shuffle=True)
work_queue.add_summary()
# create file reader
reader = tf.TextLineReader()
# get 2 records
keys, values = reader.read_up_to(work_queue.input_producer(), num_records=2)
with tf.train.MonitoredTrainingSession() as sess:
    sess.run(...)
```

### Use TableRecordReader as data sources

```
from tensorflow.python.ops.work_queue import WorkQueue

work_queue = WorkQueue(
    [odps_path1, odps_path2, odps_path3], shuffle=True, num_slices=FLAGS.num_workers * 10)
# create table reader
reader = tf.TableRecordReader()
# get 2 records
keys, values = reader.read_up_to(work_queue.input_producer(), num_records=2)
```

## 3.32 KafkaDataset

### 3.32.1 Description

1. KafkaDataset supports configuring multiple partitions and consumes kafka messages in time sequence.
2. KafkaDataset supports saving/restoring state information.

### 3.32.2 User API

```
class KafkaDataset(dataset_ops.Dataset):
    def __init__(
        self,
        topics,
        servers="localhost",
        group="",
        eof=False,
        timeout=1000,
        config_global=None,
        config_topic=None,
        message_key=False,
    )
```

- topics: A tf.string tensor containing one or more subscriptions, in the format of [topic:partition:offset:length], by default length is -1 for unlimited.

- **servers**: A list of bootstrap servers.
- **group**: The consumer group id.
- **eof**: If True, the kafka reader will stop on EOF.
- **timeout**: The timeout value for the Kafka Consumer to wait (in millisecond).
- **config\_global**: A tf.string tensor containing global configuration properties in [Key=Value] format, eg. ["enable.auto.commit=false", "heartbeat.interval.ms=2000"], please refer to 'Global configuration properties' in librdkafka doc.
- **config\_topic**: A tf.string tensor containing topic configuration properties in [Key=Value] format, eg. ["auto.offset.reset=earliest"], please refer to 'Topic configuration properties' in librdkafka doc.
- **message\_key**: If True, the kafka will output both message value and key.

### 3.32.3 Examples

```
import tensorflow as tf
from tensorflow.python.data.ops import iterator_ops

kafka_dataset = tf.data.KafkaDataset(topics=["test_1_partition:0:0:-1"],
                                     group="test_group1",
                                     timeout=100,
                                     eof=False)

iterator = iterator_ops.Iterator.from_structure(batch_dataset.output_types)
init_op = iterator.make_initializer(kafka_dataset)
get_next = iterator.get_next()
saveable_obj = tf.data.experimental.make_saveable_from_iterator(iterator)
tf.add_to_collection(tf.GraphKeys.SAVEABLE_OBJECTS, saveable_obj)
saver=tf.train.Saver()
with tf.Session() as sess:
    sess.run(init_op)
    for i in range(100):
        print("Data", sess.run(get_next))
    saver.save(sess, "ckpt/1")
```

---

## 3.33 KafkaGroupIODataset

### 3.33.1 Description

1. KafkaGroupIODataset supports configuring multiple partitions and consumes kafka messages in time sequence.
2. KafkaGroupIODataset supports load balancing within the consumer group.



### 3.33.2 User API

```
class KafkaGroupIODataset(dataset_ops.Dataset):
    def __init__(
        self,
        topics,
        group_id,
        servers,
        stream_timeout=0,
        message_poll_timeout=10000,
        configuration=None,
        internal=True,
    )
```

- **topics**: A `tf.string` tensor containing topic names in [topic] format. For example: ["topic1", "topic2"].
- **group\_id**: The id of the consumer group. For example: cgstream.
- **servers**: An optional list of bootstrap servers. For example: localhost:9092.
- **stream\_timeout**: An optional timeout duration (in milliseconds) to block until the new messages from kafka are fetched. By default it is set to 0 milliseconds and doesn't block for new messages. To block indefinitely, set it to -1.
- **message\_poll\_timeout**: An optional timeout duration (in milliseconds) after which the kafka consumer throws a timeout error while fetching a single message. This value also represents the intervals at which the kafka topic(s) are polled for new messages while using the `stream_timeout`.
- **configuration**: An optional `tf.string` tensor containing configurations in [Key=Value] format.
  - **Global configuration**: please refer to 'Global configuration properties' in librdkafka doc. Examples include ["enable.auto.commit=false", "heartbeat.interval.ms=2000"]
  - **Topic configuration**: please refer to 'Topic configuration properties' in librdkafka doc. Note all topic configurations should be prefixed with `conf.topic..` Examples include ["conf.topic.auto.offset.reset=earliest"]
  - **Reference**: <https://github.com/edenhill/librdkafka/blob/master/CONFIGURATION.md>.
- **internal**: Whether the dataset is being created from within the named scope. Default: True.

### 3.33.3 Examples

## 3.34 ParquetDataset

### 3.34.1 Description

1. ParquetDataset supports reading data from parquet files.
2. ParquetDataset supports reading parquet files from local or S3/OSS/HDFS file systems.

### 3.34.2 UserAPI

#### Environment Variable

```
# If ARROW_NUM_THREADS > 0, specified number of threads will be used.
# If ARROW_NUM_THREADS = 0, no threads will be used.
# If ARROW_NUM_THREADS < 0, all threads will be used.
os.environ['ARROW_NUM_THREADS'] = '2'
```

#### ParquetDataset API

```
class ParquetDataset(dataset_ops.DatasetV2):
    def __init__(
        self, filenames,
        batch_size=1,
        fields=None,
        partition_count=1,
        partition_index=0,
        drop_remainder=False,
        num_parallel_reads=None,
        num_sequential_reads=1):

    # Create a `ParquetDataset` from filenames dataset.
    def read_parquet(
        batch_size,
        fields=None,
        partition_count=1,
        partition_index=0,
        drop_remainder=False,
        num_parallel_reads=None,
        num_sequential_reads=1):
```

- **filenames:** the filename of parquet file, This parameter can receive the following types.
  - A 0-D or 1-D `tf.string` tensor
  - string
  - list or tuple of string
  - Dataset containing one or more filenames.
- **batch\_size:** (*Optional.*) Maxium number of samples in an output batch.
- **fields:** (*Optional.*) List of DataFrame fields.

filenames parameter type	fields parameter requirement	fields parameter type
Tensor/DataSet	required	DataFrame.Field/list or tuple of DataFrame.Field
string/list or tuple of string	optional, the default value means read all columns	DataFrame.Field/string/list or tuple of DataFrame.Field

- **partition\_count:** (*Optional.*) Count of row group partitions.

- `partition_index`: (*Optional.*) Index of row group partitions.
- `drop_remainder`: (*Optional.*) If True, only keep batches with exactly `batch_size` samples.
- `num_parallel_reads`: (*Optional.*) A `tf.int64` scalar representing the number of files to read in parallel. Defaults to reading files sequentially.
- `num_sequential_reads`: (*Optional.*) A `tf.int64` scalar representing the number of batches to read in sequential. Defaults to 1.

## DataFrame

A data frame is a table consisting of multiple named columns. A named column has a logical data type and a physical data type.

### Logical Type of DataFrame

Logical Type	Output Type
Scalar	<code>tf.Tensor/DataFrame.Value</code>
Fixed-Length List	<code>tf.Tensor/DataFrame.Value</code>
Variable-Length List	<code>tf.SparseTensor/DataFrame.Value</code>
Variable-Length Nested List	<code>tf.SparseTensor/DataFrame.Value</code>

### Physical Type of DataFrame

Category	Physical Type
Integers	<code>int64 uint64 int32 uint32 int8 uint8</code>
Numerics	<code>float64 float32 float16</code>
Text	<code>string</code>

### DataFrame API

```
class DataFrame(object):
    class Field(object):
        def __init__(self, name,
                     type=None,
                     ragged_rank=None,
                     shape=None):

        class Value(collections.namedtuple(
            'DataFrameValue', ['values', 'nested_row_splits'])):
            def to_sparse(self, name=None):

# Convert values to tensors or sparse tensors from input dataset.
def to_sparse(num_parallel_calls=None):
```

### DataFrame.Field API

- `name`: Name of column.
- `type`: data type of column, such as `tf.int64`
- `ragged_rank`: (*optional.*) Specify the number of nesting levels when column is a nested list
- `shape`: (*optional.*) Specify the shape of column when column is a fixed-length list

Attention: For fix-length list, only the shape needs to be specified.

### DataFrame.Value Conversion API (Use according to the actual situation)

Since there may be `DataFrame.Value` types in the output of `ParquetDataset` that cannot be accessed by model directly, it needs to convert the `DataFrame.Value` to `SparseTensor`. Please use the `to_sparse` API for conversion.

```
import tensorflow as tf
from tensorflow.python.data.experimental.ops import parquet_dataset_ops
from tensorflow.python.data.experimental.ops import dataframe

ds = parquet_dataset_ops.ParquetDataset(...)
ds.apply(dataframe.to_sparse())
...
```

## 3.34.3 Examples

### 1. Read from one file on local filesystem

```
import tensorflow as tf
from tensorflow.python.data.experimental.ops import parquet_dataset_ops

# Read from a parquet file.
ds = parquet_dataset_ops.ParquetDataset('/path/to/f1.parquet',
                                         batch_size=1024)

ds = ds.prefetch(4)
it = tf.data.make_one_shot_iterator(ds)
batch = it.get_next()
# {'a': tensora, 'c': tensorc}
```

### 2. Read from filenames dataset

```
import tensorflow as tf
from tensorflow.python.data.experimental.ops import parquet_dataset_ops

filenames = tf.data.Dataset.from_generator(func, tf.string, tf.TensorShape([]))
# Define data frame fields.
fields = [
    parquet_dataset_ops.DataFrame.Field('A', tf.int64),
    parquet_dataset_ops.DataFrame.Field('C', tf.int64, ragged_rank=1)]
# Read from parquet files by reading upstream filename dataset.
```

(continues on next page)

(continued from previous page)

```
ds = filenames.apply(parquet_dataset_ops.read_parquet(1024, fields=fields))
ds = ds.prefetch(4)
it = tf.data.make_one_shot_iterator(ds)
batch = it.get_next()
# {'a': tensora, 'c': tensorc}
```

### 3. Read from files on HDFS

```
import tensorflow as tf
from tensorflow.python.data.experimental.ops import parquet_dataset_ops

# Read from parquet files on remote services for selected fields.
ds = parquet_dataset_ops.ParquetDataset(
    ['hdfs://host:port/path/to/f3.parquet'],
    batch_size=1024,
    fields=['a', 'c'])
ds = ds.prefetch(4)
it = tf.data.make_one_shot_iterator(ds)
batch = it.get_next()
# {'a': tensora, 'c': tensorc}
```

## 3.35 TensorRT

TensorRT is a high-performance deep learning inference engine launched by NVIDIA. It can optimize the deep learning model into an efficient inference engine, so as to achieve fast and low-latency inference in the production environment. In order to facilitate users to use DeepRec and TensorRT at the same time, we integrate TensorRT into DeepRec. DeepRec will analyze the user's graph and cluster the subgraphs that TensorRT can recognize. This is similar to the XLA clustering graph method. For each clustering subgraph, use a TRT EngineOp drive it to execute.

### 3.35.1 Environment configuration

At present, the TensorRT environment is not installed in the image released by DeepRec for the time being, and the user needs to install it manually. The next image release will bring the installed environment. You can download TRT yourself or use the tar we offered: [TRT-8.4.2.4](#)

Unzip the tar package and copy it to a suitable location, for example: `/usr/local/TensorRT-8.4.2.4`, and then we need to set the environment variables as follows:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/TensorRT-8.4.2.4/lib
export TENSORRT_INSTALL_PATH=/usr/local/TensorRT-8.4.2.4
```

### 3.35.2 generate saved\_model

Users need to generate their own saved\_model. The document takes “model-zoo/features/embedding\_variable/wide\_and\_deep/train.py” as an example. Users first need to generate a saved model:

```
python train.py --saved_model_path=./pb
```

### 3.35.3 convert model

To convert the generated model into a TRT model, you need to use the “tensorflow/compiler/tf2tensorrt/tool/tf2trt.py” script for conversion.

```
python tf2trt.py
```

Note that there are some parameters in the tf2trt.py file that need to be configured by yourself.

```
...
if __name__ == "__main__":
    run_params = {
        # Model accuracy requirements, the default is FP32,
        # TensorRT provides FP16 quantization and INT8 quantization.
        'precision_mode': "FP32",

        # Whether to generate dynamic TRT ops that will
        # build the TRT network and engine at runtime.
        'dynamic_engine': True,

        # For INT8 quantization, TRT needs to be calibrated
        # to generate a calibration file.
        'use_calibration': False,

        # max size for the input batch.
        'max_batch_size': 1024,

        # convert model online or offline
        'convert_online': False,

        # User model use EmbeddingVariable or not.
        'use_ev': True,
    }

    # the path to load the SavedModel.
    saved_model_dir = '/model/pb'
    # The converted model save location.
    trt_saved_model_dir = './trtmodel'

    ConvertGraph(run_params, saved_model_dir, trt_saved_model_dir)
...
```

### 3.35.4 Run model

Use [DeepRec-AI/serving](#) to load new saved\_model. Note that serving needs to be recompiled, and the environment variable `export TF_NEED_TENSORRT=1` needs to be configured before this compilation, so as to ensure that the symbol table of serving so contains the TRT symbol.

## 3.36 BladeDISC

### 3.36.1 Description

BladeDISC is an end-to-end machine learning compiler open-sourced by Alibaba, which can be directly used in DeepRec.

BladeDISC project address: <https://github.com/alibaba/BladeDISC>.

At present, DeepRec and BladeDISC cannot be directly compiled from source code, and we will use this method in the future. We need to compile and generate the BladeDISC whl package, and import `blade_disc` in the user code to use. For the scenario of using C++ for serving, the serving framework needs to link the BladeDISC so. The steps are as follows.

### 3.36.2 How To Enable BladeDISC

#### 1. Compile DeepRec

For compilation instruction, please refer to [DeepRec-Compile-And-Install](#). The generated whl package will be used when compiling BladeDISC.

Note: Currently, the versions of bazel required to compile DeepRec and BladeDISC are inconsistent (this is one of the reasons why direct source code compilation is currently not possible, and we will upgrade to the same version later), so we will use the virtualenv environment to compile BladeDISC below.

#### 2. Compile BladeDISC

In the docker container that compiles DeepRec.

- Install the DeepRec whl package.
- Download BladeDISC source code.

```
git clone https://github.com/alibaba/BladeDISC.git
git checkout features/deeprec2208-cu114
git submodule update --init --recursive
```

- Configure the compilation environment.

```
# prepare venv
pip3 install virtualenv

python3 -m virtualenv /opt/venv_disc/

source /opt/venv_disc/bin/activate
```

(continues on next page)

(continued from previous page)

```
pip3 install tensorflow-1.15.5+deeprec2208-cp36-cp36m-linux_x86_64.whl
```

```
# install bazel
cd BladeDISC
apt-get update
bash ./docker/scripts/install-bazel.sh
```

- compile BladeDISC

```
# configure
./scripts/python/tao_build.py /opt/venv_disc/ --compiler-gcc default --bridge-gcc_
↪ default -s configure

# generate libtao_ops.so, path: tao/bazel-bin/libtao_ops.so
./scripts/python/tao_build.py /opt/venv_disc/ -s build_tao_bridge

# generate tao_compiler_main
# path: tf_community/bazel-bin/tensorflow/compiler/decoupling/tao_compiler_main
./scripts/python/tao_build.py /opt/venv_disc/ -s build_tao_compiler

# generate disc whl package
cp tf_community/bazel-bin/tensorflow/compiler/decoupling/tao_compiler_main tao/python/
↪ blade_disc_tf
cp tao/bazel-bin/libtao_ops.so tao/python/blade_disc_tf
cd tao
python3 setup.py bdist_wheel
```

- Install BladeDISC

```
pip install dist/blade_disc_tf1155-0.2.0-py3-none-any.whl
```

### 3. Use BladeDISC in python

In user code:

```
import blade_disc_tf as disc
disc.enable()
```

### 4. Use BladeDISC in c++ serving

The c++ serving code needs to link libtao\_ops.so, and the following two environment variables need to be set to enable disc optimization:

```
export BRIDGE_ENABLE_TAO=true
export TAO_COMPILER_PATH=/path-to/tao_compiler_main
```

Taking tensorflow\_serving as an example, we can specify the path of libtao\_ops.so (-L/xxx/xxx/mylib/) through -L at compile time, or copy libtao\_ops.so to the system lib path (for example: /usr/local/lib /), so that libtao\_ops.so can be linked to tensorflow\_serving.

Assuming that the compiled libtao\_ops.so is located at: /xxx/libtao\_ops.so, we need to modify it as follows:



```
apt-get update && apt-get install patchelf
patchelf --remove-needed libtensorflow_framework.so.1 /xxx/libtao_ops.so

# for runtime
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/xxx/
export LD_LIBRARY_PATH
```

```
apt-get update
apt-get install autotools-dev
apt-get install automake
apt-get install libtool
export TF_CUDA_COMPUTE_CAPABILITIES="7.0,7.5,8.0"
```

The modification in tensorflow\_serving is as follows: tensorflow\_serving/model\_servers/BUILD

```
cc_binary(
  name = "tensorflow_model_server_main_lib",
  ...
  deps = [
    ...
    "@org_tensorflow//tensorflow/core/platform/hadoop:hadoop_file_system",
    "@org_tensorflow//tensorflow/core/platform/s3:s3_file_system",
+    "@org_tensorflow//tensorflow/stream_executor",
+    "@org_tensorflow//tensorflow/stream_executor:stream_executor_impl",
+    "@org_tensorflow//tensorflow/stream_executor:stream_executor_internal",
+    "@org_tensorflow//tensorflow/stream_executor:stream_executor_pimpl",
+    "@org_tensorflow//tensorflow/stream_executor:kernel_spec",
+    "@org_tensorflow//tensorflow/stream_executor:kernel",
+    "@org_tensorflow//tensorflow/stream_executor:scratch_allocator",
+    "@org_tensorflow//tensorflow/stream_executor:timer",
+    "@org_tensorflow//tensorflow/stream_executor/host:host_platform",
+  ],
+  linkopts = [
+    "-ltao_ops -L/xxx/",
+    "-Wl,-no-as-needed",
+  ],
  ...
)
```

At the same time, because the above BUILD file relies on stream\_executor, and the current visibility of stream\_executor is “friends”, not “public”, here we need to modify the DeepRec file ./tensorflow/stream\_executor/BUILD referenced by tensorflow\_serving as follows:

```
package(
-   default_visibility = [":friends"],
+   default_visibility = ["/visibility:public"],
  licenses = ["notice"], # Apache 2.0
)
```

When BladeDISC is compiled with DeepRec, GLIBCXX\_USE\_CXX11\_ABI=1 is used by default, while tensorflow\_serving uses GLIBCXX\_USE\_CXX11\_ABI=0 by default, so both sides need to be unified. This document takes modifying the .bazelrc file of tensorflow\_serving as an example:

```
- build --cxxopt=-D_GLIBCXX_USE_CXX11_ABI=0
+ build --cxxopt=-D_GLIBCXX_USE_CXX11_ABI=1
```

tensorflow\_serving compilation script:

```
bazel build -c opt --config=cuda tensorflow_serving/...
```

tensorflow\_serving compilation: [tfs compilation](#)

## 3.37 XLA

XLA (Accelerated Linear Algebra) is a domain-specific linear algebra compiler that speeds up the running of Tensor-Flow models, possibly with no source code changes at all. The usage is the same as that of native Tensorflow.

### 3.37.1 Enable XLA for full graph

```
sess_config = tf.ConfigProto()
...
sess_config.graph_options.optimizer_options.global_jit_level = tf.OptimizerOptions.ON_1
...

with tf.train.MonitoredTrainingSession(
    ...
    config=sess_config) as sess:
    ...
```

### 3.37.2 Enable XLA for subgraph

```
def dnn(self, dnn_input, dnn_hidden_units=None, layer_name=''):
    # Add the following code to limit part of the graph to use XLA to compile.
    #
    jit_scope = tf.contrib.compiler.jit.experimental_jit_scope
    with jit_scope():
        for layer_id, num_hidden_units in enumerate(dnn_hidden_units):
            with tf.variable_scope(layer_name + "_%d" % layer_id,
                                   partitioner=self.dense_layer_partitioner,
                                   reuse=tf.AUTO_REUSE) as dnn_layer_scope:
                dnn_input = tf.layers.dense(dnn_input,
                                             units=num_hidden_units,
                                             activation=tf.nn.relu,
                                             name=dnn_layer_scope)

            if self.use_bn:
                dnn_input = tf.layers.batch_normalization(
                    dnn_input, training=self.is_training, trainable=True)
                add_layer_summary(dnn_input, dnn_layer_scope.name)

    return dnn_input
```

## 3.38 Processor

### 3.38.1 Introduction

DeepRec Serving Processor is a library for online high-performance services. It is based on DeepRec and support the following functions:

- Supports automatic discovery and import of full models;
- Support incremental model updates, reduce the time consumption caused by loading the model, and make the model online more real-time;
- Supports asynchronous update of models to reduce online serving performance jitter;
- Support model rollback to different versions;
- Support model WarmUp, avoiding the problem of a slow start when loading the model for the first time;
- Support model local storage (memory + Pmem + SSD) and distributed storage services (multi-node shared storage model parameters), etc.
- Compatible with native Tensorflow, it can also serve normally for graphs trained with native Tensorflow.
- The easy-to-use API interface makes it easier for users to use.

The Processor is an independent .so file that users can easily link to their own Serving RPC framework.

### 3.38.2 Compile

Compile details [compile processor](#), we will get “**libserving\_processor.so**” after compiling.

### 3.38.3 Usage

Users can use it in two ways:

First, directly [dlopen](#) in the user framework code to load the symbols in so.

Second, use the header files “**serving/processor/serving/processor.h**” and “**libserving\_processor.so**”.

**Attention:** If you are not using DeepRec docker, then some additional .so dependencies may be required, including: libiomp5.so, libmklml\_intel.so, libstdc++.so.6. You can download them directly from [here](#).

#### C API

Processor provides the following C API interfaces, and users need to call the following interfaces in their Serving framework.

##### 1) initialize

```
void* initialize(const char* model_entry, const char* model_config, int* state);
```

##### Args:

model\_entry: By default, the string “” is passed (note that it is not NULL).

model\_config: The json content read from the configuration file.

state: Return to the user the status of the Serving framework, 0 is normal, -1 is abnormal.

**Return value:** Return a pointer, which is the `model_buf` argument in the process function below. This argument needs to be passed in each time the process function is called.

**Usage:** The initialize function is called once when the Serving RPC framework starts. The framework needs to save the returned pointer, and this parameter needs to be passed in when the process function is subsequently called.

**Example:**

```
const char* entry = "xxx";
const char* config = "json configs";
int state = -1;
void* model = initialize(entry, config, &state);
```

## 2) process

```
int process(void* model_buf, const void* input_data, int input_size, void** output_data,
↳int* output_size);
```

**Args:**

`model_buf`: The returned pointer value of initialize function.

`input_data`: The user request serialized into a byte stream by protobuf, the protobuf format is shown in PredictRequest “data format” below.

`input_size`: The size of the request.

`output_data`: The result returned by the prediction is a serialized byte stream in the protobuf format. For the protobuf format, see PredictResponse “data format” below. (Note: The returned buffer is allocated on the heap memory, and the user framework needs to be responsible for reclaiming the memory, otherwise there will be a memory leak.)

`output_size`: The size of the response.

**Return value:** Return status code, 200 means OK, 500 means service error.

**Usage:** The user Serving framework receives the request sent by the Client. If the requested data format is valid for the Processor (see “Data Format” below), it can directly call the Process function to perform prediction. If the data format is invalid, it needs to be converted into the format required by the Processor (see “Data Format” below) and then call the Process function.

**Example:**

```
void* model = initialize(xx, xx, xx);
...
char* input_data = "xxx";
int input_size = 3;
void* output_data = nullptr;
int output_size = 0;
int state = process(model, (void*)input_data, input_size, &output_data, &output_size);
```

## 3) get\_serving\_model\_info

```
int get_serving_model_info(void* model_buf, void** output_data, int* output_size);
```

**Args:**

`model_buf`: The returned pointer value of initialize function.

`output_data`: The returned result is a serialized byte stream in protobuf format. For the protobuf format, see Serving-ModelInfo “data format” below. (Note: The returned buffer is allocated on the heap memory, and the user framework needs to be responsible for reclaiming the memory, otherwise there will be a memory leak.)

output\_size: The size of output\_data.

**Return value** Return status code, 200 means OK, 500 means service error.

**Usage:** Users can call this API query when they need information about the model currently being served.

**Example:**

```
void* model = initialize(xx, xx, xx);
...
void* output_data = nullptr;
int output_size = 0;
int state = get_serving_model_info(model, &output_data, &output_size);
```

### data format

The Request, Response and other data formats required by the Processor are as follows. Here we use Protobuf as the data storage format. Consistent with “**serving/processor/serving/predict.proto**” under DeepRec. Protobuf is the abbreviation of Protocol Buffers. It is a data description language used to describe a portable and efficient structured data storage format. Protobuf can be used for structured data serialization or serialization. Simply put, data can be parsed from one language to another. After Java’s Protobuf data is serialized, it can be parsed in C++ as it is, which facilitates data exchange in various scenarios. The user needs to encapsulate the data into the following format on the client side, or convert it to this format before call Process function.

```
enum ArrayDataType {
    // Not a legal value for DataType. Used to indicate a DataType field
    // has not been set.
    DT_INVALID = 0;

    // Data types that all computation devices are expected to be
    // capable to support.
    DT_FLOAT = 1;
    DT_DOUBLE = 2;
    DT_INT32 = 3;
    DT_UINT8 = 4;
    DT_INT16 = 5;
    DT_INT8 = 6;
    DT_STRING = 7;
    DT_COMPLEX64 = 8; // Single-precision complex
    DT_INT64 = 9;
    DT_BOOL = 10;
    DT_QINT8 = 11;    // Quantized int8
    DT_QUINT8 = 12;   // Quantized uint8
    DT_QINT32 = 13;   // Quantized int32
    DT_BFLOAT16 = 14; // Float32 truncated to 16 bits. Only for cast ops.
    DT_QINT16 = 15;   // Quantized int16
    DT_QUINT16 = 16;  // Quantized uint16
    DT_UINT16 = 17;
    DT_COMPLEX128 = 18; // Double-precision complex
    DT_HALF = 19;
    DT_RESOURCE = 20;
    DT_VARIANT = 21; // Arbitrary C++ data types
}
```

(continues on next page)

(continued from previous page)

```

// Dimensions of an array
message ArrayShape {
  repeated int64 dim = 1 [packed = true];
}

// Protocol buffer representing an array
message ArrayProto {
  // Data Type.
  ArrayDataType dtype = 1;

  // Shape of the array.
  ArrayShape array_shape = 2;

  // DT_FLOAT.
  repeated float float_val = 3 [packed = true];

  // DT_DOUBLE.
  repeated double double_val = 4 [packed = true];

  // DT_INT32, DT_INT16, DT_INT8, DT_UINT8.
  repeated int32 int_val = 5 [packed = true];

  // DT_STRING.
  repeated bytes string_val = 6;

  // DT_INT64.
  repeated int64 int64_val = 7 [packed = true];

  // DT_BOOL.
  repeated bool bool_val = 8 [packed = true];
}

// PredictRequest specifies which TensorFlow model to run, as well as
// how inputs are mapped to tensors and how outputs are filtered before
// returning to user.
message PredictRequest {
  // A named signature to evaluate. If unspecified, the default signature
// will be used
  string signature_name = 1;

  // Input tensors.
  // Names of input tensor are alias names. The mapping from aliases to real
// input tensor names is expected to be stored as named generic signature
// under the key "inputs" in the model export.
  // Each alias listed in a generic signature named "inputs" should be provided
// exactly once in order to run the prediction.
  map<string, ArrayProto> inputs = 2;

  // Output filter.
  // Names specified are alias names. The mapping from aliases to real output
// tensor names is expected to be stored as named generic signature under
// the key "outputs" in the model export.

```

(continues on next page)

(continued from previous page)

```

// Only tensors specified here will be run/fetched and returned, with the
// exception that when none is specified, all tensors specified in the
// named signature will be run/fetched and returned.
repeated string output_filter = 3;
}

// Response for PredictRequest on successful run.
message PredictResponse {
  // Output tensors.
  map<string, ArrayProto> outputs = 1;
}

// Response for current serving model info
message ServingModelInfo {
  string model_path = 1;
  // Add other info here
}

```

The user generates the class function of the corresponding language according to the above xxx.proto file, for example, xxx.pb.cc and xxx.pb.h are generated in C++, and the corresponding java files are generated in Java.

In the above code, PredictRequest is the request data structure, and PredictResponse is the response data structure returned to the user. About fields in PredictRequest and PredictResponse, we can obtain some information from **saved\_model.pb/saved\_model.pbtxt**, assuming that the content of **saved\_model.pbtxt** is as follows:

```

signature_def {
  key: "serving_default"
  value {
    inputs {
      key: "input1"
      value {
        name: "input1:0"
        dtype: DT_DOUBLE
        tensor_shape {
          dim {
            size: -1
          }
        }
      }
    }
    inputs {
      key: "input2"
      value {
        name: "input2:0"
        dtype: DT_INT32
        tensor_shape {
          dim {
            size: -1
          }
        }
      }
    }
    outputs {

```

(continues on next page)

(continued from previous page)

```

    key: "probabilities"
    value {
      name: "prediction:0"
      dtype: DT_FLOAT
      tensor_shape {
        dim {
          size: -1
        }
      }
    }
  }
  method_name: "tensorflow/serving/predict"
}

```

**For PredictRequest:**

- 1) string signature\_name: User-specified signature, which can be seen from saved\_model.pbtxt, here is 'serving\_default'.
- 2) map<string, ArrayProto> inputs: Feeds, which is a map type data, key is input name, which is string type; value is the tensor (dense tensor) corresponding to this input, which is ArrayProto type, and ArrayProto is a pb array, see the definition above for details. Suppose there is an age input, and the tensor is ArrayProto t, and its value is [10], then inputs["age"] = t; if there are multiple inputs, just add them in. If there are two inputs in the above example, then the input is: {"input1:0":tensor1, "input2:0":tensor2}.
- 3) repeated string output\_filter: Fetches, the name of the fetch tensor returned by predict is required. In the above example, there is an output, and the output\_filter is: {"prediction:0"}.

**For PredictResponse:**

map<string, ArrayProto> outputs: It is a map structure, the key is the names specified in the output\_filter in PredictRequest, and the value is the returned tensor.

**Configure file**

As mentioned above, the initialize function needs to be called during initialization:

```
void* initialize(const char* model_entry, const char* model_config, int* state);
```

The second argument "**model\_config**" of the initialize function is a content in json format, which has the following fields. For details, please refer to the open source code file: [serving/processor/serving/model\\_config.cc](#)

```

{
# Enable 'session group', and set the number of sessions in the group to this value.
"session_num": 2,

# If session group is used, it indicates how each session run selects the session in the
↪group to execute sess_run.
# The methods include:
# "MOD": According to the thread number of the request, the session num is moduloed to
↪obtain the session num serving the current request.
# "RR": Polling to select the session in the group for service.
# default is 'RR',

```

(continues on next page)



(continued from previous page)

```

"select_session_policy": "MOD",

# When using session group, true means that each session
# in the group has an independent inter/intra thread pool.
"use_per_session_threads": false,

# Users can set different cpu cores for each session in the session group,
# The format is as follows:
# "0-10;11-20" means that two sessions are bound to 0~10cores and 11~20cores.
↳respectively.
# or
# "0,1,2,3;4,5,6,7" means that two sessions are bound to 0~3cores and 4~7cores.
↳respectively.
# Different session cpu cores are separated by ';'.
"cpusets": "123;4-6",

# Users can set which physical GPUs are used by the current
# session group through options, as shown below to use GPU 0 and GPU 2.
# It should be noted that the GPU number here may not
# the same as the number from nvidia-smi.
# For example, if the user sets CUDA_VISIBLE_DEVICES=3,2,1,0,
# then the numbers 0,1,2,3 seen in deeprec correspond to physical GPUs 3,2,1,0.
# The user does not need to care about the specific GPU,
# but only needs to know the number of visible GPUs in deeprec,
# and then make corresponding settings.
"gpu_ids_list": "0,2",

# whether to use multi-stream In GPU tasks
"use_multi_stream": false,

# Whether to enable device placement optimization in GPU tasks
"enable_device_placement_optimization": false,

# Whether to execute Session run in a single thread
"enable_inline_execute": false,

# The default value is 4 (parameters are related to MKL performance and need to be
↳debugged)
"omp_num_threads": 4,

# The default value is 0 (parameters are related to MKL performance and need to be
↳debugged)
"kmp_blocktime": 0,

# Argument required to load the model, 'local' or 'redis'
# Represents loading the model into memory (local hybrid storage) and loading model
↳parameters into redis respectively.
"feature_store_type": "local",

# [required when feature_store_type is 'redis']
"redis_url": "redis_url",

```

(continues on next page)

(continued from previous page)

```

# [required when feature_store_type is 'redis']
"redis_password": "redis_password",

# [required when feature_store_type is 'redis']
# Redis read thread number
"read_thread_num": 4,

# [required when feature_store_type is 'redis'],
# Redis updating thread number
"update_thread_num": 1,

# Default serialization uses protobuf (reserved argument)
"serialize_protocol": "protobuf",

# DeepRec inter thread num
"inter_op_parallelism_threads": 10,

# DeepRec intra thread num
"intra_op_parallelism_threads": 10,

# Model hot update uses Session's own inter thread pool by default.
# If the user sets this parameter, an additional model update 'inter' thread pool will
↳ be created.
# The parameter value indicates the number of threads.
"model_update_inter_threads": 4,

# Model hot update uses Session's own inter thread pool by default.
# If the user sets this parameter, an additional model update 'intra' thread pool will
↳ be created.
# The parameter value indicates the number of threads.
"model_update_intra_threads": 4,

# Default value 1 (reserved parameter)
"init_timeout_minutes": 1,

# signature_name, which can be obtained from saved model.pbtxt.
"signature_name": "serving_default",

# warmup file, not used means no warmup.
"warmup_file_name": "warm_up.bin",

# The storage of user model files, currently supports local/oss/hdfs
# local: "/root/a/b/c"
# oss: "oss://bucket/a/b/c"
# hdfs: "hdfs://a/b/c"
"model_store_type": "oss",

# If model_store_type is oss, then set checkpoint_dir and savedmodel_dir to oss path.
# If it is local or hdfs, then set the corresponding path.
# checkpoint_dir requires specifying the parent directory of a specific checkpoint dir,
# For example: oss://mybucket/test/ckpt_parent_dir/, then multiple versions of
↳ checkpoints

```

(continues on next page)

(continued from previous page)

```

# are allowed in this directory, such as: checkpoint1/, checkpoint2/, checkpoint3/ ...
# For each version of checkpoint is the standard Tensorflow checkpoint directory.
↳structure.
# savedmodel_dir will not be updated unless the graph changes, and currently needs to be.
↳manually restarted to update.
"checkpoint_dir": "oss://mybucket/test/ckpt_parent_dir/",
"savedmodel_dir": "oss://mybucket/test/savedmodel/1616466677/",

# If oss is used, set the oss-related access id and access key below
"oss_endpoint": "oss_endpoint",
"oss_access_id": "oss_access_id",
"oss_access_key": "oss_access_key",

# If you need to print the timeline, the timeline_start_step parameter indicates
# that the timeline will start from the set number of steps
"timeline_start_step": 1,

# timeline_interval_step indicates how many steps to print a new timeline at intervals
"timeline_interval_step": 2,

# timeline_trace_count indicates how many timelines need to be collected in total
"timeline_trace_count": 3,

# timeline save location, support oss and local
# local: "/root/timeline/"
"timeline_path": "oss://mybucket/timeline/",

# EmbeddingVariable storage configuration
# 0: Use the deeperc default configuration, 1: DRAM single-level storage, 12:
↳DRAM+SSDHASH multi-level storage
# Default value: 0
"ev_storage_type": 12,

# Set multi-level storage path, if multi-level storage is enable
"ev_storage_path": "/ssd/1/",

# The size of each level of storage in multi-level storage
"ev_storage_size": [1024, 1024]
}

```

### Export saved\_model

If the user enables the incremental\_ckpt function in training, then Processor can use the incremental\_ckpt to update the service during serving, thus ensuring the real-time performance of the service model.

About exporting the saved\_model, users can use several different APIs, including directly using the low-level API SavedModelBuilder, or using the high-level API such as estimator. For tasks that use incremental\_ckpt function, the save\_incr\_model switch needs to be turned on when exporting the model, so that the corresponding incremental restore subgraph can be found in the saved model.

## SavedModelBuilder

If the user exports the saved\_model by splicing the low-level APIs, it is necessary to ensure that the save\_incr\_model parameter is set to true (the default is false) when building the SavedModelBuilder.

```
class SavedModelBuilder(_SavedModelBuilder):
    def __init__(self, export_dir, save_incr_model=False):
        super(SavedModelBuilder, self).__init__(export_dir=export_dir, save_incr_model=save_incr_model)
    ...
```

## Estimator

If the user uses estimator, the parameter save\_incr\_model needs to be set to True when calling estimator.export\_saved\_model,

```
estimator.export_saved_model(
    export_dir_base,
    serving_input_receiver_fn,
    ...
    save_incr_model=True)
```

## Model path configuration

In Processor, users need to provide checkpoint and saved\_model paths, and processor reads meta graph information from saved\_model, including signature, input, output and other information. The model parameters need to be read from the checkpoint, because the current incremental update depends on the checkpoint instead of the saved model. During the serving process, when the checkpoint is updated and the processor finds a new version of the model in the specified model directory, it will automatically load the latest model. The saved model is generally not updated unless the graph changes. If it does change, a new processor instance needs to be restarted.

The user provided files are as follows:

checkpoint:

```
/a/b/c/checkpoint_parent_dir/
|  _ _ checkpoint_1/...
|  _ _ checkpoint_2/...
|  _ _ checkpoint_3/
|          |  _ _ checkpoint
|          |  _ _ graph.pbtxt
|          |  _ _ model.ckpt-0.index
|          |  _ _ model.ckpt-0.meta
|          |  _ _ model.ckpt-0.data-00000-of-00001
|          |  _ _ .incremental_ckpt/...
```

The above checkpoint\_1~checkpoint\_3 are a complete model directory, including full model and incremental model.

saved\_model:

```

/a/b/c/saved_model/
|  _ _ saved_model.pb
|  _ _ variables

```

Taking the above as an example, in the configuration file, “checkpoint\_dir” is set to “/a/b/c/checkpoint\_parent\_dir/”, and “savedmodel\_dir” is set to “/a/b/c/saved\_model/”

## Warmup

The default Model Warmup in EAS is executed when the eas task is started. For the ODL processor, because the model will be automatically updated during the serving process, the Warmup is also required for the new model, so we provide Warmup function in serving.

```

{
  ...
  "model_config": {
    ...
    "warmup_file_name": "/xxx/xxx/warm_up.bin",
    ...
  }
  ...
}

```

Processor currently does not support downloading warmup files, and users have two ways to provide warmup files.

1.EASmount oss, example as follows,

```

"storage": [
  {
    "mount_path": "/data_oss",
    "oss": {
      "endpoint": "oss-cn-shanghai-internal.aliyuncs.com",
      "path": "oss://bucket/"
    }
  }
]

```

Mount oss://bucket/ on the local /data\_oss directory, assuming that the location of the warmup file in oss is “oss://bucket/1/warmup.bin”, then in the configuration file, you need to set the warmup\_file\_name to “/data\_oss/1/warmup.bin”

2.If the user does not mount oss, another way is to use EAS to download the warmup file, the configuration is as follows,

```

{
  ...
  "model_config": {
    ...
    "warmup_file_name": "/home/admin/docker_ml/workspace/model/warm_up.bin",
    ...
  }
  "warm_up_data_path": "oss://my_oss/1/warm_up.bin",
  ...
}

```

The warmup parameter “warm\_up\_data\_path” of EAS needs to be configured: “oss://my\_oss/1/warm\_up.bin” so that the EAS framework will download this file, and the download path is “/home/admin/docker\_ml/workspace/model/warm\_up. bin” (different versions of eas may change, you need to consult EAS developer), the user can set the warmup\_file\_name to “/home/admin/docker\_ml/workspace/model/warm\_up.bin”, so that the processor can also find the warmup\_file for warmup.

### 3.38.4 End2End example

Details see: `serving/processor/tests/end2end/README` A complete end-to-end example is provided here.

### 3.38.5 Timeline collection

By configuring timeline-related parameters in the config file, the corresponding timeline files can be obtained. These files are in binary format and cannot be displayed directly in `chrome://tracing`. Users need to go to the DeepRec directory (usually `/root/.cache/bazel/_bazel_root/`) to find the `config_pb2.py` file which is required by `gen_timeline.py` script, and put it in the `serving/tools/timeline` directory, execute `gen_timeline.py` in this directory to generate the timeline that `chrome://tracing` can display.

```
# usage:
python gen_timeline.py timeline_file my_timeline.json
```

## 3.39 SessionGroup

### 3.39.1 Introduction

In the current recommendation inference scenario, the user always used the `tensorflow_serving` framework or EAS+Processor which is a serving framework in Alibaba. There is usually only one session in the process of these frameworks, and a single session makes it impossible to efficiently utilize resources such as CPU and GPU. Of course, users can perform tasks in a multi-instance mode (multi-process), but this method cannot share the Variables, resulting in a large amount of memory usage, and each Instance loads the model once, seriously affecting resource usage and model loading efficiency.

By using SessionGroup, it can solve the problem of large memory usage but low model CPU usage, greatly improve resource utilization, and greatly improve QPS under the premise of ensuring latency. In addition, multiple sessions in SessionGroup can also be executed concurrently in GPU scenarios, which greatly improves the utilization efficiency of GPUs.

### 3.39.2 Usage

If users use `tensorflow_serving` for services, they can use the code we provide: [DeepRec-AI/serving](#), here has already provided the function of accessing SessionGroup. You can also use the [Processor](#) code provided by us. Processor does not provide an RPC service framework, you can use our RPC framework [PAI-EAS](#) or yours.

## 1.Processor + EAS

### CPU Task

If users use session\_group on EAS processor, they only need to add the following fields in the configuration file:

```
"model_config": {
  "session_num": 2,
  "use_per_session_threads": true,
  ...
}
```

### GPU Task

For GPU tasks, the following configurations are required:

```
"model_config": {
  "session_num": 2,
  "use_per_session_threads": true,
  "gpu_ids_list": "0,2",
  ...
}
```

More parameters see: [processor configuration parameters](#)

## 2.Tensorflow serving

Our offered tensorflow\_serving now do serving via SavedModelBundle. We already support SessionGroup in tensorflow\_serving, related code modification reference: [SessionGroup](#), it is recommended to use the tensorflow\_serving code provided by us directly.

tensorflow\_serving repo that supports SessionGroup: [DeepRec-AI/serving](#)

Compilation documentation see: [TFServing compilation](#)

### CPU Task

Run command:

```
bazel-bin/tensorflow_serving/model_servers/tensorflow_model_server --tensorflow_intra_op_
↪parallelism=16 --tensorflow_inter_op_parallelism=16 --use_per_session_threads=true --
↪session_num_per_group=4 --model_base_path=/xxx/pb
```

Related Args:

```
session_num_per_group: Indicates how many sessions are created in the SessionGroup.

use_per_session_threads: If it is true, it means that each session uses an independent_
↪thread pool to reduce the interference between sessions. It is recommended to_
↪configure it as true. The thread pool size of each session is controlled by tensorflow_
↪intra_op_parallelism and tensorflow_inter_op_parallelism.
```

Users can specify the CPU cores for each session in the SessionGroup. The default function is disabled. There are two ways to enable it:

```
1.User manually sets environment variables:
SESSION_GROUP_CPUSET="2-4;5-7;8-10;11-13"
Or
SESSION_GROUP_CPUSET="2,3,4;5,6,7;8,9,10;11,12,13"
This indicates that there are 4 sessions, and each session is executed on the specified
CPU.
session0: 2 3 4
session1: 5 6 7
session2: 8 9 10
session3: 11 12 13

2.User needs to set SET_SESSION_THREAD_POOL_AFFINITY=1 if they does not set the
environment variable SESSION_GROUP_CPUSET.
DeepRec will detect which CPUs can be allocated, and then allocate CPU cores to
different sessions according to the distribution of CPUs on NUMA nodes.
```

These options can be used in GPU task.

### GPU Task

In Inference scenarios, users often use GPUs for online services to improve computing efficiency and reduce latency. One problem that may be encountered here is that the online GPU utilization rate is low, resulting in a waste of resources. Then, to make good use of GPU resources, we use Multi-streams to process requests, which greatly improves QPS while ensuring latency. In the GPU scenario, using session group will use multi-stream by default, that is, each session uses an independent stream.

At present inference scenarios, the multi-streams function is bound to the SessionGroup function. For the usage of SessionGroup, see the previous link. In the future, we will directly support the multi-streams function on DirectSession.

The specific usage is the same as that of SessionGroup, and the following modifications need to be made on this basis.

### 1.Docker startup configuration

This function uses GPU MPS (Multi-Process Service) optimization(optional, recommended to enable), which requires the background MPS service process to be started in the docker container.

```
nvidia-cuda-mps-control -d
```

### 2.Startup command

Currently taking Tensorflow\_serving as an example (it will be necessary to add other framework usage methods later), the following parameters need to be added when starting the server.

```
CUDA_VISIBLE_DEVICES=0 ENABLE_MPS=1 CONTEXTS_COUNT_PER_GPU=4 MERGE_COMPUTE_COPY_
STREAM=1 PER_SESSION_HOSTALLOC=1 bazel-bin/tensorflow_serving/model_servers/tensorflow_
model_server --tensorflow_intra_op_parallelism=8 --tensorflow_inter_op_parallelism=8 --
use_per_session_threads=true --session_num_per_group=4 --allow_gpu_mem_growth=true --
model_base_path=/xx/xx/pb/
```

(continues on next page)



(continued from previous page)

```

ENABLE_MPS=1: Turn on MPS (it is generally recommended to turn on).
CONTEXTS_COUNT_PER_GPU=4: Configure cuda contexts count for each physical GPU, the
↪ default is 4.
MERGE_COMPUTE_COPY_STREAM: The calculation operator and the copy operator use the
                           same stream to reduce waiting between different streams.
PER_SESSION_HOSTALLOC=1: Each session uses an independent gpu host allocator.

use_per_session_threads=true: Each session configures the thread pool separately.
session_num_per_group=4: Indicates the number of sessions configured by the session
↪ group.

```

### 3.Multi-GPU

If the user does not specify `CUDA_VISIBLE_DEVICES=0` and there are multiple GPUs on the machine, the session group will use all GPUs by default. Assuming there are 2 GPUs, and `session_num_per_group=4` is set, then the session group will create 4 streams on each GPU, because currently a stream corresponds to a session, so there are a total of  $2*4=8$  sessions in the current session group. The model parameters required by these sessions on the CPU are all shared. For the model parameters of the place on the GPU, if the stream associated with the session is on the same GPU, then the GPU parameters are shared between these sessions, otherwise the sessions don't share GPU parameters.

Users can specify which physical GPUs are assigned to a session group,

```
--gpu_ids_list=0,2
```

The option above indicate that GPUs 0 and 2 are assigned to the current session group. It should be noted that the GPU number here does not correspond to the number seen by `nvidia-smi`, but the number seen by `deeprec`.

For example, suppose there are 4 GPUs on the physical machine, and the numbers are 0, 1, 2, 3. If the user set nothing, then the numbers 0, 1, 2, 3 and the physical GPU numbers seen in `deeprec` are the same. If the user sets `CUDA_VISIBLE_DEVICES=3,2,1,0`, then the physical GPU numbers corresponding to the numbers 0,1,2,3 seen in `deeprec` are 3,2,1,0 respectively.

The corresponding relationship above does not affect the actual use. Users only need to care about how many physical GPUs `deeprec` can actually see. Assuming that 3 GPUs can be seen, then the visible numbers of `deeprec` are 0, 1, and 2.

The option `--gpu_ids_list=0,2` means that the user can use GPUs 0 and 2. If the number of GPUs visible to the process is less than 3 (0,1,2), an error will be reported.

Startup command:

```

ENABLE_MPS=1 CONTEXTS_COUNT_PER_GPU=4 bazel-bin/tensorflow_serving/model_servers/
↪ tensorflow_model_server --tensorflow_intra_op_parallelism=8 --tensorflow_inter_op_
↪ parallelism=8 --use_per_session_threads=true --session_num_per_group=4 --allow_gpu_
↪ mem_growth=true --gpu_ids_list=0,2 --model_base_path=/xx/xx/pb/

```

For a detailed explanation of the above environment variables, see: [Startup parameters](#)

#### 4. Best practice for using MPS with multiple GPU

There may be multiple GPUs on the user machine, and generally only one GPU Device is required for each serving instance, so the user may start multiple different serving instances on the physical machine. There are some issues to be aware of when using MPS in this situation, as follows:

- 1) The MPS daemon process needs to be started on the physical machine so that all tasks in docker can be managed by the MPS background process.

```
nvidia-cuda-mps-control -d
```

- 2) When starting docker, you need to add ‘--ipc=host’ to ensure that the process in docker is visible to the MPS daemon process. At the same time, for each docker, it is recommended to mount the specified GPU Device, as follows:

```
sudo docker run -it --name docker_name --ipc=host --net=host --gpus='"device=0"' docker_
↪image bash
```

In this way, only one GPU will be visible in docker, and the logical number is 0, then the script can be executed as follows:

```
CUDA_VISIBLE_DEVICES=0 test.py
Or
test.py
```

If docker mounts all GPU Devices, then when executing the script, you need to manually specify the visible GPU device to achieve the effect of resource isolation.

```
sudo docker run -it --name docker_name --ipc=host --net=host --gpus=all docker_image bash

docker0 task:
CUDA_VISIBLE_DEVICES=0 test.py

docker1 task:
CUDA_VISIBLE_DEVICES=1 test.py
```

The tensorflow\_serving code modified by DeepRec is as follows: [TF serving](#)

#### 3. Use user-defined framework

If users want to implement the session group function into their own framework, they can refer to the code implementation in the processor below.

##### Create SessionGroup

If you manually create a serving framework implemented by Session::Run, then modify NewSession API in the serving framework to NewSessionGroup. session\_num specifies how many sessions are created in the SessionGroup, user can judge how many sessions need to be created by evaluating the CPU utilization of the current single session. For example, if the current maximum CPU utilization of a single session is 20%, it is recommended that users configure 5 sessions.

```
TF_RETURN_IF_ERROR(NewSessionGroup(*session_options_,
    session_group, session_num));
TF_RETURN_IF_ERROR((*session_group)->Create(meta_graph_def_.graph_def()));
```

Reference Code: [Processor](#)

### SessionGroup Run API

The Session::Run API used by users can be directly replaced by SessionGroup::Run.

```
status = session_group->Run(run_options, req.inputs,
    req.output_tensor_names, {}, &resp.outputs, &run_metadata);
```

Reference Code: [Processor](#)

## 3.39.3 Multi-model service

### TF Serving

SessionGroup supports multi-model services, which have been supported on [TF\\_serving](#). For multi-model services, users can configure independent parameters for each model service, including the number of sessions in session groups in different model services, specifying GPUs, thread pools, etc., so as to isolate frameworks and resources.

For GPU tasks, the session group can specify one or more GPUs, so users need to pay attention to the division of GPU resources when starting multi-model tasks.

The command to start the multi-model service is as follows:

```
ENABLE_MPS=1 CONTEXTS_COUNT_PER_GPU=4 MERGE_COMPUTE_COPY_STREAM=1 PER_SESSION_
↪HOSTALLOC=1 bazel-bin/tensorflow_serving/model_servers/tensorflow_model_server --rest_
↪api_port=8888 --use_session_group=true --model_config_file=/data/workspace/serving-
↪model/multi_wdl_model/models.config --platform_config_file=/data/workspace/serving-
↪model/multi_wdl_model/platform_config_file
```

For a detailed explanation of the above environment variables, see: [Startup parameters](#)

The following are the two most important configuration files in the command,

Assuming that there are 4 GPU devices on the machine, the configuration is as follows:

model\_config\_file:

```
model_config_list:{
  config:{
    name:"pb1",
    base_path:"/data/workspace/serving-model/multi_wdl_model/pb1",
    model_platform:"tensorflow",
    model_id: 0
  },
  config:{
    name:"pb2",
    base_path:"/data/workspace/serving-model/multi_wdl_model/pb2",
    model_platform:"tensorflow",
    model_id: 1
  },
}
```

For each model service, a corresponding model\_config\_file is required

- name: indicates the name of the model service. When requesting access from the client side, the corresponding service name request.model\_spec.name = 'pb1' needs to be filled.
- base\_path: indicates the path where the model is located.
- model\_platform: default value 'tensorflow'.
- model\_id: Give each model service a number, starting from 0.

platform\_config\_file:

```
platform_configs {
  key: "tensorflow"
  value {
    source_adapter_config {
      [type.googleapis.com/tensorflow.serving.SavedModelBundleV2SourceAdapterConfig] {
        legacy_config {
          model_session_config {
            session_config {
              gpu_options {
                allow_growth: true
              }
              intra_op_parallelism_threads: 8
              inter_op_parallelism_threads: 8
              use_per_session_threads: true
              use_per_session_stream: true
            }
            session_num: 2
            cpusets: "1,2;5,6"
            gpu_ids: [0,1]
          }
          model_session_config {
            session_config {
              gpu_options {
                allow_growth: true
              }
              intra_op_parallelism_threads: 16
              inter_op_parallelism_threads: 16
              use_per_session_threads: true
              use_per_session_stream: true
            }
            session_num: 2
            cpusets: "20,21;23,24;26,27;29,30"
            gpu_ids: [2,3]
          }
        }
      }
    }
  }
}
```

The key is the same as the model\_platform field above, and the default value is 'tensorFlow'. For each model service, a model\_session\_config needs to be configured, including some configurations of the session. model\_session\_config is finally an array, then model\_session\_config[0] represents the configuration of model\_0 service, and so on.

Note the gpu\_ids parameter above, indicating which GPUs are used by each session group. Assume here that there are 4 GPUs on the machine, then use gpu-0, gpu1 and gpu-2, gpu-3 on the two session groups respectively. Special

attention needs to be paid. At this time, the total number of sessions in the session group is `session_num*gpu_ids.size`, that is, 4 sessions.

Server example:

```
CUDA_VISIBLE_DEVICES=1,3 ENABLE_MPS=1 MERGE_COMPUTE_COPY_STREAM=1 PER_SESSION_
↪HOSTALLOC=1 bazel-bin/tensorflow_serving/model_servers/tensorflow_model_server --rest_
↪api_port=8888 --use_session_group=true --model_config_file=/xxx/model_config_file --
↪platform_config_file=/xxx/platform_config_file
```

For a detailed explanation of the above environment variables, see: [Startup parameters](#)

Client example:

```
...
request = predict_pb2.PredictRequest()
request.model_spec.name = 'pb2' # set model name here, like 'pb1', 'pb2' ...
request.model_spec.signature_name = 'serving_default'
...
```

## EAS+Processor

The distribution of multi-model requests requires the support of the EAS framework. We can configure different cpu resources, thread pools, etc. for different model services. The configuration files are as follows:

```
{
  "platform": "local",
  "engine": "python",
  "language_type": "PYTHON",
  "name": "pttest",
  "models": [
    {
      "model_path": "https://tf115test.oss-cn-hangzhou.aliyuncs.com/test/lib-serving_
↪processor_1226_debug3.tar.gz",
      "model_entry": "",
      "name": "model1",
      "processor": "tensorflow",
      "uncompress": true,
      "model_config": {
        "session_num": 2,
        "use_per_session_threads": true,
        "cpusets": "1,2,3;4,5,6",
        "omp_num_threads": 24,
        "kmp_blocktime": 0,
        "feature_store_type": "memory",
        "serialize_protocol": "protobuf",
        "inter_op_parallelism_threads": 24,
        "intra_op_parallelism_threads": 24,
        "init_timeout_minutes": 1,
        "signature_name": "serving_default",
        "model_store_type": "local",
        "checkpoint_dir": "/data/workspace/ckpt/",
        "savedmodel_dir": "/data/workspace/pb/",
        "oss_access_id": "",
```

(continues on next page)

(continued from previous page)

```

        "oss_access_key": "",
        "oss_endpoint": "oss-cn-shanghai.aliyuncs.com"
    }
},
{
    "model_path": "https://tf115test.oss-cn-hangzhou.aliyuncs.com/test/lib-serving_
↪processor_1226_debug3.tar.gz",
    "model_entry": "",
    "name": "model2",
    "processor": "tensorflow",
    "uncompress": true,
    "model_config": {
        "session_num": 4,
        "use_per_session_threads": true,
        "cpusets": "7-9;10-12;13-15;16-18",
        "omp_num_threads": 24,
        "kmp_blocktime": 0,
        "feature_store_type": "memory",
        "serialize_protocol": "protobuf",
        "inter_op_parallelism_threads": 24,
        "intra_op_parallelism_threads": 24,
        "init_timeout_minutes": 1,
        "signature_name": "serving_default",
        "model_store_type": "local",
        "checkpoint_dir": "/data/workspace/ckpt2/",
        "savedmodel_dir": "/data/workspace/pb2/",
        "oss_access_id": "",
        "oss_access_key": "",
        "oss_endpoint": "oss-cn-shanghai.aliyuncs.com"
    }
}
],
"processors": [
    {
        "name": "tensorflow",
        "processor_path": "https://tf115test.oss-cn-hangzhou.aliyuncs.com/test/lib-serving_
↪processor_1226_debug3.tar.gz",
        "processor_entry": "lib-serving_processor.so",
        "processor_type": "cpp"
    }
],
"metadata": {
    "cpu": 32,
    "eas": {
        "enabled_model_verification": false,
        "scheduler": {
            "enable_cpuset": false
        }
    }
},
"gpu": 0,
"instance": 1,
"memory": 40960,

```

(continues on next page)

(continued from previous page)

```

    "rpc":{
        "io_threads":10,
        "worker_threads":20,
        "enable_jemalloc":true
    },
}

```

The difference between the multi-model service configuration file and the single-model configuration file is that the “models” and “processors” fields are added.

The “processors” field is a list. Users can configure multiple processors. In the “models” field, different processors can be configured for different model services.

The “models” field is a list. Users can configure fields separately for multiple model services. Each model service has a separate configuration, mainly the “model\_config” field. For more detailed field introductions, see:

Example:

```

client = PredictClient('127.0.0.1:8080', 'pttest/model1')
#client = PredictClient('127.0.0.1:8080', 'pttest/model2')
client.init()

pb_string = open("./warm_up.bin", "rb").read()
request = TFRequest()
request.request_data.ParseFromString(pb_string)
request.add_fetch("dinfm/din_out/Sigmoid:0")

resp = client.predict(request)

```

When building ‘PredictClient’, you need to add the model name to the url, such as “pttest/model1”.

## 3.40 Device Placement Optimization

### 3.40.1 Background

In the online GPU inference task of the sparse model, only part of the embedding layer operators are placed on the GPU, which leads to a large amount of data copying between the CPU and GPU when the inference task executes the Embedding layer operators. As a result, the performance improvement brought by GPU computing acceleration is difficult to offset the overhead brought by memory copy, slowing down the inference task. Although it can be solved by implementing the GPU version of related operators, some operators still have problems such as low parallelism and some calculations must be performed on the CPU, resulting in low execution efficiency of the GPU version of the operator, or the GPU version of the operator is difficult to implement. Therefore, users can place the operators of the Embedding layer on the CPU, to reduce memory copies and improve performance.

We propose the Device Placement optimization function, which can automatically place the Embedding Layer on the CPU, thus solving the performance degradation problem caused by the large number of data copy operations between CPU and GPU during the execution of the Embedding Layer.

### 3.40.2 Description

The Device Placement optimization function can automatically identify operators in the Embedding layer and place them on the CPU. This function only changes the actual computation graph, not the GraphDef.

### 3.40.3 How to use

In user C++ code:

```
tensorflow::SessionOptions session_options;
session_options.config.mutable_graph_options()->mutable_optimizer_options()->set_device_
  placement_optimization(true);
```

If we use `tensorflow-serving` offered by DeepRec, we can enable the optimization via `--enable_device_placement_optimization=true`

And when we use DeepRec `processor`, we should add `enable_device_placement_optimization` field in the json configuration file, as follows:

```
{
  "model_entry": "",
  "processor_path": "...",
  "processor_entry": "lib-serving_processor.so",
  "processor_type": "cpp",
  "model_config": {
    ...
    "enable_device_placement_optimization": true,
    ...
  },
  ...
}
```