Tutorial: TF-Ranking for sparse features

This tutorial is an end-to-end walkthrough of training a TensorFlow Ranking (TF-Ranking) neural network model which incorporates sparse textual features.

TF-Ranking is a library for solving large scale ranking problems using deep learning. TF-Ranking can handle heterogeneous dense and sparse features, and scales up to millions of data points. For more details, please read the technical paper published on arXiv (https://arxiv.org/abs/1812.00073).

Run in Google Colab (https://colab.research.google.com/github/tensorflow/ranking/blob/master/tensorflow_ranking/examples/handling_sparse_features.ipynb)

View source on Gi (https://github.con

Motivation

This tutorial demonstrates how to build ranking estimators over sparse features, such as textual data. Textual data is prevalent in several settings for ranking, and plays a significant role is relevance judgment by a user.

In Search and Question Answering tasks, queries and document titles are examples of textual information. In Recommendation task, the titles of the items and their descriptions contain textual information.

Hence it is important for LTR (Learning-to-Rank) models to effectively incorporate textual features.

Data Formats and a Ranking Task

Data Formats for Ranking

For representing ranking data, protobuffers (https://developers.google.com/protocol-buffers/) are extensible structures suitable for storing data in a serialized format, either locally or in a distributed manner.

Ranking usually consists of features corresponding to each of the examples being sorted. In addition, features related to query, user or session are also useful for ranking. We refer to these as context features, as these are independent of the examples.

We use the popular tf.Example (https://www.tensorflow.org/tutorials/load_data/tf_records) proto to represent the features for context, and each of the examples. We create a new format for ranking data, **Example in Example** (EIE), to store context as a serialized tf.Example proto and the list of examples to be ranked as a list of serialized tf.Example protos.

ANTIQUE: A Question Answering Dataset

ANTIQUE (http://hamedz.ir/resources/) is a publicly available dataset for open-domain non-factoid question answering, collected over Yahoo! answers.

Each question has a list of answers, whose relevance are graded on a scale of 1-5.

This dataset is a suitable one for learning-to-rank scenario. The dataset is split into 2206 queries for training and 200 queries for testing. For more details, please read the tehcnical paper on arXiv (https://arxiv.org/pdf/1905.08957.pdf).

Download training, test data and vocabulary file.

In [0]:

```
!wget -O "/tmp/vocab.txt" "http://ciir.cs.umass.edu/downloads/Antique/tf-ranking/v
ocab.txt"
!wget -O "/tmp/train.tfrecords" "http://ciir.cs.umass.edu/downloads/Antique/tf-ran
king/train.tfrecords"
!wget -O "/tmp/test.tfrecords" "http://ciir.cs.umass.edu/downloads/Antique/tf-rank
ing/test.tfrecords"
```

Dependencies and Global Variables

Let us start by importing libraries that will be used throughout this Notebook. We also enable the "eager execution" mode for convenience and demonstration purposes.

In [0]:
```
import six
import os
import numpy as np

try:
  import tensorflow as tf
except ImportError:
  print('Installing TensorFlow.  This will take a minute, ignore the warnings.')
  !pip install -q tensorflow
  import tensorflow as tf

try:
  import tensorflow_ranking as tfr
except ImportError:
    !pip install -q tensorflow_ranking
    import tensorflow_ranking as tfr

tf.enable_eager_execution()
tf.executing_eagerly()
tf.set_random_seed(1234)
tf.logging.set_verbosity(tf.logging.INFO)
```

Here we define the train and test paths, along with model hyperparameters.

In [0]:
```
# Store the paths to files containing training and test instances.
_TRAIN_DATA_PATH = "/tmp/train.tfrecords"
_TEST_DATA_PATH = "/tmp/test.tfrecords"

# Store the vocabulary path for query and document tokens.
_VOCAB_PATH = "/tmp/vocab.txt"

# The maximum number of documents per query in the dataset.
# Document lists are apdded or truncated to this size.
_LIST_SIZE = 50

# The document relevance label.
_LABEL_FEATURE = "relevance"

# Padding labels are set negative so that the corresponding examples can be
# ignored in loss and metrics.
_PADDING_LABEL = -1

# Learning rate for optimizer.
_LEARNING_RATE = 0.05

# Parameters to the scoring function.
_BATCH_SIZE = 32
_HIDDEN_LAYER_DIMS = ["64", "32", "16"]
_DROPOUT_RATE = 0.8
_GROUP_SIZE = 1  # Pointwise scoring.

# Location of model directory and number of training steps.
_MODEL_DIR = "/tmp/ranking_model_dir"
_NUM_TRAIN_STEPS = 15 * 1000
```

Components of a Ranking Estimator

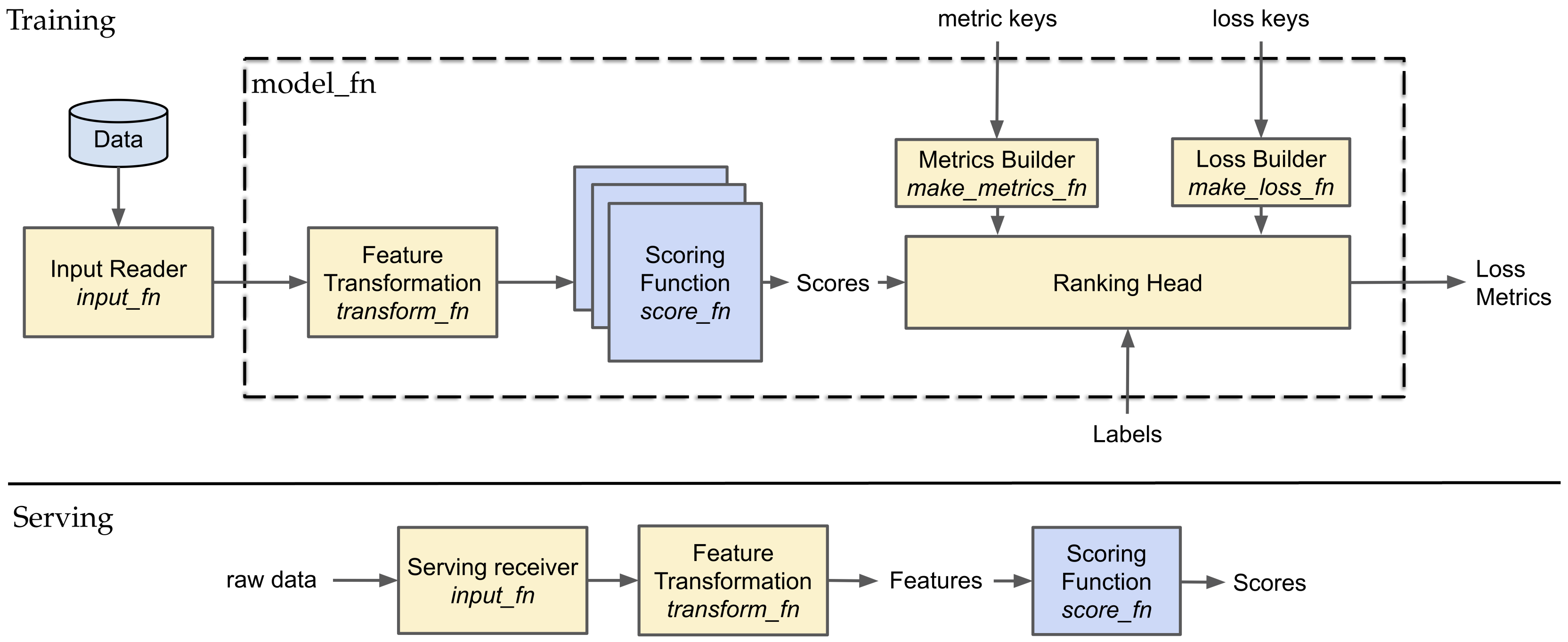
The overall components of a Ranking Estimator are shown below.

The key components of the library are:

1. Input Reader
2. Tranform Function
3. Scoring Function
4. Ranking Losses
5. Ranking Metrics
6. Ranking Head
7. Model Builder

These are described in more details in the following sections.

TensorFlow Ranking Architecture



Specifying Features via Feature Columns

Feature Columns (https://www.tensorflow.org/guide/feature_columns) are TensorFlow abstractions that are used to capture rich information about each feature. It allows for easy transformations for a diverse range of raw features and for interfacing with Estimators.

Consistent with our input formats for ranking, such as EIE format, we create feature columns for context features and example features.

In [0]:

```
_EMBEDDING_DIMENSION = 20

def context_feature_columns():
  """Returns context feature names to column definitions."""
  sparse_column = tf.feature_column.categorical_column_with_vocabulary_file(
      key="query_tokens",
      vocabulary_file=_VOCAB_PATH)
  query_embedding_column = tf.feature_column.embedding_column(
      sparse_column, _EMBEDDING_DIMENSION)
  return {"query_tokens": query_embedding_column}

def example_feature_columns():
  """Returns the example feature columns."""
  sparse_column = tf.feature_column.categorical_column_with_vocabulary_file(
      key="document_tokens",
      vocabulary_file=_VOCAB_PATH)
  document_embedding_column = tf.feature_column.embedding_column(
      sparse_column, _EMBEDDING_DIMENSION)
  return {"document_tokens": document_embedding_column}
```

Reading Input Data using *input_fn*

The input reader reads in data from persistent storage to produce raw dense and sparse tensors of appropriate type for each feature. Example features are represented by 3-D tensors (where dimensions correspond to queries, examples and feature values). Context features are represented by 2-D tensors (where dimensions correspond to queries and feature values).

In [0]:

```
def input_fn(path, num_epochs=None):
  context_feature_spec = tf.feature_column.make_parse_example_spec(
        context_feature_columns().values())
  label_column = tf.feature_column.numeric_column(
        _LABEL_FEATURE, dtype=tf.int64, default_value=_PADDING_LABEL)
  example_feature_spec = tf.feature_column.make_parse_example_spec(
        list(example_feature_columns().values()) + [label_column])
  dataset = tfr.data.build_ranking_dataset(
        file_pattern=path,
        data_format=tfr.data.EIE,
        batch_size=_BATCH_SIZE,
        list_size=_LIST_SIZE,
        context_feature_spec=context_feature_spec,
        example_feature_spec=example_feature_spec,
        reader=tf.data.TFRecordDataset,
        shuffle=False,
        num_epochs=num_epochs)
  features = tf.data.make_one_shot_iterator(dataset).get_next()
  label = tf.squeeze(features.pop(_LABEL_FEATURE), axis=2)
  label = tf.cast(label, tf.float32)
  return features, label
```

Feature Transformations with *transform_fn*

The transform function takes in the raw dense or sparse features from the input reader, applies suitable transformations to return dense representations for each faeture. This is important before passing these features to a neural network, as neural networks layers usually take dense features as inputs.

The transform function handles any custom feature transformations defined by the user. For handling sparse features, like text data, we provide an easy utlity to create shared embeddings, based on the feature columns.

In [0]:

```
def make_transform_fn():
  def _transform_fn(features, mode):
    """Defines transform_fn."""
    example_name = next(six.iterkeys(example_feature_columns()))
    input_size = tf.shape(input=features[example_name])[1]
    context_features, example_features = tfr.feature.encode_listwise_features(
        features=features,
        input_size=input_size,
        context_feature_columns=context_feature_columns(),
        example_feature_columns=example_feature_columns(),
        mode=mode,
        scope="transform_layer")

    return context_features, example_features
  return _transform_fn
```

Feature Interactions using *scoring_fn*

Next, we turn to the scoring function which is arguably at the heart of a TF Ranking model. The idea is to compute a relevance score for a (set of) query-document pair(s). The TF-Ranking model will use training data to learn this function.

Here we formulate a scoring function using a feed forward network. The function takes the features of a single example (i.e., query-document pair) and produces a relevance score.

In [0]:
```python
def make_score_fn():
  """Returns a scoring function to build `EstimatorSpec`."""

  def _score_fn(context_features, group_features, mode, params, config):
    """Defines the network to score a group of documents."""
    with tf.compat.v1.name_scope("input_layer"):
      context_input = [
          tf.compat.v1.layers.flatten(context_features[name])
          for name in sorted(context_feature_columns())
      ]
      group_input = [
          tf.compat.v1.layers.flatten(group_features[name])
          for name in sorted(example_feature_columns())
      ]
      input_layer = tf.concat(context_input + group_input, 1)

    is_training = (mode == tf.estimator.ModeKeys.TRAIN)
    cur_layer = input_layer
    cur_layer = tf.compat.v1.layers.batch_normalization(
      cur_layer,
      training=is_training,
      momentum=0.99)

    for i, layer_width in enumerate(int(d) for d in _HIDDEN_LAYER_DIMS):
      cur_layer = tf.compat.v1.layers.dense(cur_layer, units=layer_width)
      cur_layer = tf.compat.v1.layers.batch_normalization(
        cur_layer,
        training=is_training,
        momentum=0.99)
      cur_layer = tf.nn.relu(cur_layer)
      cur_layer = tf.compat.v1.layers.dropout(
          inputs=cur_layer, rate=_DROPOUT_RATE, training=is_training)
    logits = tf.compat.v1.layers.dense(cur_layer, units=_GROUP_SIZE)
    return logits

  return _score_fn
```

Losses, Metrics and Ranking Head

Evaluation Metrics

We have provided an implementation of several popular Information Retrieval evaluation metrics in the TF Ranking library, which are shown here (https://github.com/tensorflow/ranking/blob/d8c2e2e64a92923f1448cf5302c92a80bb469a
The user can also define a custom evaluation metric, as shown in the description below.

Ranking Losses

We provide several popular ranking loss functions as part of the library, which are shown here (https://github.com/tensorflow/ranking/blob/d8c2e2e64a92923f1448cf5302c92a80bb469a

The user can also define a custom loss function, similar to ones in tfr.losses.

In [0]:
```
# Define a loss function. To find a complete list of available
# loss functions or to learn how to add your own custom function
# please refer to the tensorflow_ranking.losses module.

_LOSS = tfr.losses.RankingLossKey.APPROX_NDCG_LOSS
loss_fn = tfr.losses.make_loss_fn(_LOSS)
```

Ranking Head

In the Estimator workflow, Head is an abstraction that encapsulates losses and corresponding metrics. Head easily interfaces with the Estimator, needing the user to define a scoring function and specify losses and metric computation.

```
In [0]: optimizer = tf.compat.v1.train.AdagradOptimizer(
            learning_rate=_LEARNING_RATE)

        def _train_op_fn(loss):
          """Defines train op used in ranking head."""
          update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
          minimize_op = optimizer.minimize(
              loss=loss, global_step=tf.compat.v1.train.get_global_step())
          train_op = tf.group([update_ops, minimize_op])
          return train_op

        ranking_head = tfr.head.create_ranking_head(
              loss_fn=loss_fn,
              eval_metric_fns=eval_metric_fns(),
              train_op_fn=_train_op_fn)
```

Putting It All Together in a Model Builder

We are now ready to put all of the components above together and create an `Estimator` that can be used to train and evaluate a model.

In [0]:
```
model_fn = tfr.model.make_groupwise_ranking_fn(
          group_score_fn=make_score_fn(),
          transform_fn=make_transform_fn(),
          group_size=_GROUP_SIZE,
          ranking_head=ranking_head)
```

In [0]:
```
def train_and_eval_fn():
  """Train and eval function used by `tf.estimator.train_and_evaluate`."""
  run_config = tf.estimator.RunConfig(
      save_checkpoints_steps=1000)
  ranker = tf.estimator.Estimator(
      model_fn=model_fn,
      model_dir=_MODEL_DIR,
      config=run_config)

  train_input_fn = lambda: input_fn(_TRAIN_DATA_PATH)
  eval_input_fn = lambda: input_fn(_TEST_DATA_PATH, num_epochs=1)

  train_spec = tf.estimator.TrainSpec(
      input_fn=train_input_fn, max_steps=_NUM_TRAIN_STEPS)
  eval_spec =  tf.estimator.EvalSpec(
          name="eval",
          input_fn=eval_input_fn,
          throttle_secs=15)
  return (ranker, train_spec, eval_spec)
```

Train and evaluate the ranker

In [0]:
```
! rm -rf "/tmp/ranking_model_dir"  # Clean up the model directory.
ranker, train_spec, eval_spec = train_and_eval_fn()
tf.estimator.train_and_evaluate(ranker, train_spec, eval_spec)
```

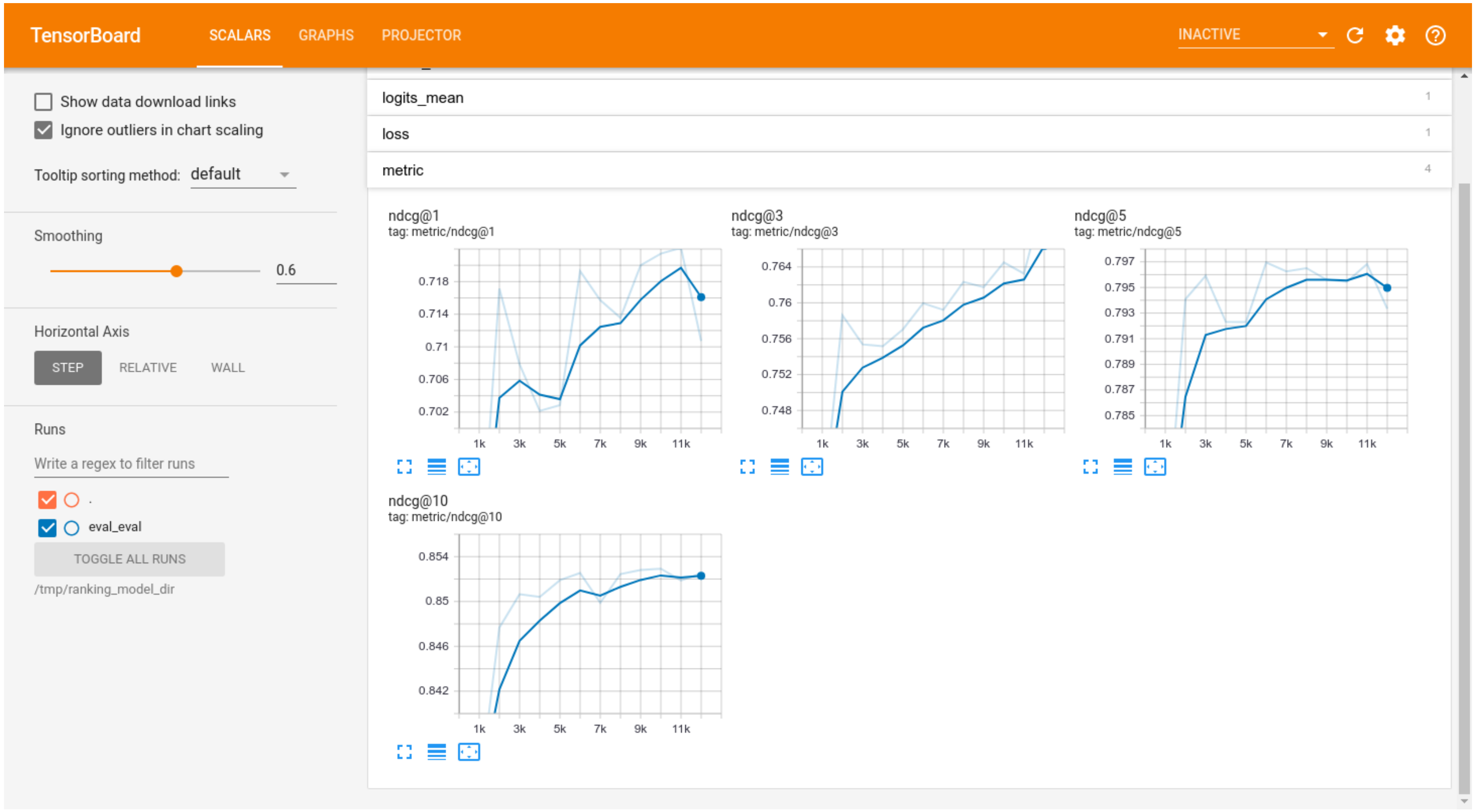
Finally, let us evaluate our model on the test set.

In [0]:
```
ranker.evaluate(input_fn=lambda: input_fn(_TEST_DATA_PATH, num_epochs=1))
```

Launch TensorBoard

```
In [0]: %load_ext tensorboard
        %tensorboard --logdir="/tmp/ranking_model_dir" --port 12345
```

A sample tensorboard output is shown here, with the ranking metrics.



TensorBoard
SCALARS
GRAPHS
PROJECTOR
INACTIVE
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
.
eval_eval
TOGGLE ALL RUNS
/tmp/ranking_model_dir
logits_mean 1
loss 1
metric 4
ndcg@1
tag: metric/ndcg@1
ndcg@3
tag: metric/ndcg@3
ndcg@5
tag: metric/ndcg@5
ndcg@10
tag: metric/ndcg@10