Training on Cloud MLE

The following code example will train the processed datasets on Google Cloud MLE. At this point, change the Notebook runtime type to Python 3.0.

- Configure GCP project.

```
# configure GCP project - update with your parameters
project_id = 'ekabasandbox'
bucket_name = 'superconductor'
region = 'us-central1'
tf_version = '1.8'

import os
os.environ['bucket_name'] = bucket_name
os.environ['tf_version'] = tf_version
os.environ['project_id'] = project_id
os.environ['region'] = region
```

- Create directory "trainer".

```
# create directory trainer
import os
try:
    os.makedirs('./trainer')
    print('directory created')
except OSError:
    print('could not create directory')
```

- Create file __init__.py.

```
%%writefile trainer/__init__.py
```

- Create the trainer file task.py. Replace the bucket name with your values.

```
%%writefile trainer/task.py
import argparse
import json
import os
import tensorflow as tf
from tensorflow.contrib.training.python.training import hparam

import trainer.model as model

def _get_session_config_from_env_var():
    """Returns a tf.ConfigProto instance that has appropriate
    device_filters set.
    """

    tf_config = json.loads(os.environ.get('TF_CONFIG', '{}'))

    if (tf_config and 'task' in tf_config and 'type' in tf_
    config['task'] and
       'index' in tf_config['task']):
        # Master should only communicate with itself and ps
        if tf_config['task']['type'] == 'master':
            return tf.ConfigProto(device_filters=['/job:ps', '/
            job:master'])
        # Worker should only communicate with itself and ps
        elif tf_config['task']['type'] == 'worker':
            return tf.ConfigProto(device_filters=[
                '/job:ps',
                '/job:worker/task:%d' % tf_config['task']['index']
            ])
    return None

def train_and_evaluate(hparams):
    """Run the training and evaluate using the high level API."""

    train_input = lambda: model.input_fn(
        tf.gfile.Glob(hparams.train_files),
        num_epochs=hparams.num_epochs,
        batch_size=hparams.train_batch_size
    )

    # Don't shuffle evaluation data
    eval_input = lambda: model.input_fn(
        tf.gfile.Glob(hparams.eval_files),
            batch_size=hparams.eval_batch_size,
            shuffle=False
        )

        train_spec = tf.estimator.TrainSpec(
            train_input, max_steps=hparams.train_steps)

        exporter = tf.estimator.FinalExporter(
            'superconductor', model.SERVING_FUNCTIONS[hparams.export_
            format])
        eval_spec = tf.estimator.EvalSpec(
            eval_input,
            steps=hparams.eval_steps,
            exporters=[exporter],
            name='superconductor-eval')

        run_config = tf.estimator.RunConfig(
            session_config=_get_session_config_from_env_var())
        run_config = run_config.replace(model_dir=hparams.job_dir)
        print('Model dir %s' % run_config.model_dir)
        estimator = model.build_estimator(
            learning_rate=hparams.learning_rate,
            # Construct layers sizes with exponential decay
            hidden_units=[
                max(2, int(hparams.first_layer_size * hparams.scale_
                factor**i))
                for i in range(hparams.num_layers)
            ],
            config=run_config,
            output_dir=hparams.output_dir)

        tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)

    if __name__ == '__main__':
        parser = argparse.ArgumentParser()
        # Input Arguments
        parser.add_argument(
            '--train-files',
        help='GCS file or local paths to training data',
        nargs='+',
        # update the bucket name
        default='gs://{}/preproc_csv/data/{}*{}*'.format('super
        conductor', tf.estimator.ModeKeys.TRAIN, 'of'))
    parser.add_argument(
        '--eval-files',
        help='GCS file or local paths to evaluation data',
        nargs='+',
        # update the bucket name
        default='gs://{}/preproc_csv/data/{}*{}*'.format('super
        conductor', tf.estimator.ModeKeys.EVAL, 'of'))
    parser.add_argument(
        '--job-dir',
        help='GCS location to write checkpoints and export models',
        default='/tmp/superconductor-estimator')
    parser.add_argument(
        '--num-epochs',
        help="""\
        Maximum number of training data epochs on which to train.
        If both --max-steps and --num-epochs are specified,
        the training job will run for --max-steps or --num-epochs,
        whichever occurs first. If unspecified will run for
        --max-steps.\
        """,
        type=int)
    parser.add_argument(
        '--train-batch-size',
        help='Batch size for training steps',
        type=int,
        default=20)
    parser.add_argument(
        '--eval-batch-size',
        help='Batch size for evaluation steps',
        type=int,
        default=20)
    parser.add_argument(
        '--learning-rate',
        help='The training learning rate',
        default=1e-4,
        type=float)
    parser.add_argument(
        '--first-layer-size',
        help='Number of nodes in the first layer of the DNN',
        default=256,
        type=int)
    parser.add_argument(
        '--num-layers', help='Number of layers in the DNN',
        default=3, type=int)
    parser.add_argument(
        '--scale-factor',
        help='How quickly should the size of the layers in the DNN
        decay',
        default=0.7,
        type=float)
    parser.add_argument(
        '--train-steps',
        help="""\
        Steps to run the training job for. If --num-epochs is not
        specified,
        this must be. Otherwise the training job will run
        indefinitely.\
        """,
        default=100,
        type=int)
    parser.add_argument(
        '--eval-steps',
        help='Number of steps to run evalution for at each
        checkpoint',
        default=100,
        type=int)
```

```
parser.add_argument(
    '--export-format',
    help='The input format of the exported SavedModel binary',
    choices=['JSON', 'CSV', 'EXAMPLE'],
    default='CSV')
parser.add_argument(
    '--output-dir',
    help='Location of the exported model',
    nargs='+')
parser.add_argument(
    '--verbosity',
    choices=['DEBUG', 'ERROR', 'FATAL', 'INFO', 'WARN'],
    default='INFO')

args, _ = parser.parse_known_args()

# Set python level verbosity
tf.logging.set_verbosity(args.verbosity)
# Set C++ Graph Execution level verbosity
os.environ['TF_CPP_MIN_LOG_LEVEL'] = str(
    tf.logging.__dict__[args.verbosity] / 10)

# Run the training job
hparams = hparam.HParams(**args.__dict__)
train_and_evaluate(hparams)
```

- Create the file model.py that contains the model code.

```
%%writefile trainer/model.py
import six

import tensorflow as tf
from tensorflow.python.estimator.model_fn import ModeKeys as Modes

# Define the format of your input data including unused columns.
CSV_COLUMNS = [
    'number_of_elements', 'mean_atomic_mass', 'entropy_atomic_mass',
   'wtd_entropy_atomic_mass', 'range_atomic_mass',
    'wtd_range_atomic_mass', 'mean_fie', 'wtd_mean_fie',
    'wtd_entropy_fie', 'range_fie', 'wtd_range_fie',
    'mean_atomic_radius', 'wtd_mean_atomic_radius',
    'range_atomic_radius', 'wtd_range_atomic_radius', 'mean_
    Density',
    'entropy_Density', 'wtd_entropy_Density', 'range_Density',
    'wtd_range_Density', 'mean_ElectronAffinity',
    'wtd_entropy_ElectronAffinity', 'range_ElectronAffinity',
    'wtd_range_ElectronAffinity', 'mean_FusionHeat', 'gmean_
    FusionHeat',
    'entropy_FusionHeat', 'wtd_entropy_FusionHeat', 'range_
    FusionHeat',
    'wtd_range_FusionHeat', 'mean_ThermalConductivity',
    'wtd_mean_ThermalConductivity', 'gmean_ThermalConductivity',
    'entropy_ThermalConductivity', 'wtd_entropy_
    ThermalConductivity',
    'range_ThermalConductivity', 'wtd_range_ThermalConductivity',
    'mean_Valence', 'wtd_mean_Valence', 'range_Valence',
    'wtd_range_Valence', 'wtd_std_Valence', 'critical_temp'
]

CSV_COLUMN_DEFAULTS = [[0.0] for i in range(0, len(CSV_COLUMNS))]
LABEL_COLUMN = 'critical_temp'

# Define the initial ingestion of each feature used by your model.
# Additionally, provide metadata about the feature.
INPUT_COLUMNS = [tf.feature_column.numeric_column(i) for i in CSV_
COLUMNS[:-1]]

UNUSED_COLUMNS = set(CSV_COLUMNS) - {col.name for col in INPUT_
COLUMNS} - \
    {LABEL_COLUMN}

def build_estimator(config, output_dir, hidden_units=None,
learning_rate=None):
    """
    Deep NN Regression model.
```

```
Args:
    config: (tf.contrib.learn.RunConfig) defining the runtime
    environment for
      the estimator (including model_dir).
    hidden_units: [int], the layer sizes of the DNN (input
    layer first)
    learning_rate: (int), the learning rate for the optimizer.
Returns:
    A DNNRegressor
"""
(number_of_elements,mean_atomic_mass,entropy_atomic_mass,wtd_
entropy_atomic_mass, \
  range_atomic_mass,wtd_range_atomic_mass,mean_fie,wtd_mean_
  fie,wtd_entropy_fie,range_fie,\
  wtd_range_fie,mean_atomic_radius,wtd_mean_atomic_
  radius,range_atomic_radius,wtd_range_atomic_radius,\
  mean_Density,entropy_Density,wtd_entropy_Density,range_
  Density,wtd_range_Density,mean_ElectronAffinity,\
  wtd_entropy_ElectronAffinity,range_ElectronAffinity,wtd_
  range_ElectronAffinity,mean_FusionHeat,\
  gmean_FusionHeat,entropy_FusionHeat,wtd_entropy_
  FusionHeat,range_FusionHeat,wtd_range_FusionHeat,\
  mean_ThermalConductivity,wtd_mean_ThermalConductivity,gmean_
  ThermalConductivity,entropy_ThermalConductivity,\
  wtd_entropy_ThermalConductivity,range_
  ThermalConductivity,wtd_range_ThermalConductivity,mean_
  Valence,\
  wtd_mean_Valence,range_Valence,wtd_range_Valence,wtd_std_
  Valence) = INPUT_COLUMNS

columns = [number_of_elements,mean_atomic_mass,entropy_atomic_
mass,wtd_entropy_atomic_mass, \
  range_atomic_mass,wtd_range_atomic_mass,mean_fie,wtd_mean_
  fie,wtd_entropy_fie,range_fie,\
  wtd_range_fie,mean_atomic_radius,wtd_mean_atomic_
  radius,range_atomic_radius,wtd_range_atomic_radius,\
          mean_Density,entropy_Density,wtd_entropy_Density,range_
          Density,wtd_range_Density,mean_ElectronAffinity,\
          wtd_entropy_ElectronAffinity,range_ElectronAffinity,wtd_
          range_ElectronAffinity,mean_FusionHeat,\
          gmean_FusionHeat,entropy_FusionHeat,wtd_entropy_FusionHeat,
          range_FusionHeat,wtd_range_FusionHeat,\
          mean_ThermalConductivity,wtd_mean_ThermalConductivity,
          gmean_ThermalConductivity,entropy_ThermalConductivity,\
          wtd_entropy_ThermalConductivity,range_ThermalConductivity,
          wtd_range_ThermalConductivity,mean_Valence,\
          wtd_mean_Valence,range_Valence,wtd_range_Valence,wtd_std_
          Valence]

        estimator = tf.estimator.DNNRegressor(
          model_dir=output_dir,
          config=config,
          feature_columns=columns,
          hidden_units=hidden_units or [256, 128, 64],
          optimizer=tf.train.AdamOptimizer(learning_rate)
        )

        # add extra evaluation metric for hyperparameter tuning
        estimator = tf.contrib.estimator.add_metrics(estimator, add_
        eval_metrics)
        return estimator

    def add_eval_metrics(labels, predictions):
        pred_values = predictions['predictions']
        return {
            'rmse': tf.metrics.root_mean_squared_error(labels,
            pred_values)
        }

    # [START serving-function]

    def csv_serving_input_fn():
        """Build the serving inputs."""
        csv_row = tf.placeholder(shape=[None], dtype=tf.string)
    features = _decode_csv(csv_row)
    # Ignore label column
    features.pop(LABEL_COLUMN)
    return tf.estimator.export.ServingInputReceiver(features,
                                              {'csv_row': csv_row})
def example_serving_input_fn():
    """Build the serving inputs."""
    example_bytestring = tf.placeholder(
      shape=[None],
      dtype=tf.string,
    )
    features = tf.parse_example(
      example_bytestring,
      tf.feature_column.make_parse_example_spec(INPUT_COLUMNS))
    return tf.estimator.export.ServingInputReceiver(
      features, {'example_proto': example_bytestring})

def json_serving_input_fn():
    """Build the serving inputs."""
    inputs = {}
    for feat in INPUT_COLUMNS:
        inputs[feat.name] = tf.placeholder(shape=[None],
        dtype=feat.dtype)

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

# [END serving-function]

SERVING_FUNCTIONS = {
  'JSON': json_serving_input_fn,
  'EXAMPLE': example_serving_input_fn,
  'CSV': csv_serving_input_fn
}

def _decode_csv(line):
    """Takes the string input tensor and returns a dict of rank-2
    tensors."""
```

```
        # Takes a rank-1 tensor and converts it into rank-2 tensor
        row_columns = tf.expand_dims(line, -1)
        columns = tf.decode_csv(row_columns, record_defaults=CSV_
        COLUMN_DEFAULTS)
        features = dict(zip(CSV_COLUMNS, columns))

        # Remove unused columns
        for col in UNUSED_COLUMNS:
            features.pop(col)
        return features

    def input_fn(filenames, num_epochs=None, shuffle=True, skip_
    header_lines=1, batch_size=200):
        """Generates features and labels for training or evaluation.
        This uses the input pipeline based approach using file name queue
        to read data so that entire data is not loaded in memory.
        Args:
          filenames: [str] A List of CSV file(s) to read data from.
          num_epochs: (int) how many times through to read the data.
          If None will loop through data indefinitely
          shuffle: (bool) whether or not to randomize the order of
          data. Controls randomization of both file order and line
          order within files.
          skip_header_lines: (int) set to non-zero in order to skip
          header lines in CSV files.
          batch_size: (int) First dimension size of the Tensors
          returned by input_fn
        Returns:
          A (features, indices) tuple where features is a dictionary of
            Tensors, and indices is a single Tensor of label indices.
        """
        dataset = tf.data.TextLineDataset(filenames).skip(skip_header_
        lines).map(
          _decode_csv)
    if shuffle:
        dataset = dataset.shuffle(buffer_size=batch_size * 10)
    iterator = dataset.repeat(num_epochs).batch(
        batch_size).make_one_shot_iterator()
    features = iterator.get_next()
    return features, features.pop(LABEL_COLUMN)
```

- Create the hyper-parameter config file.

```
%%writefile hptuning_config.yaml
trainingInput:
  hyperparameters:
    hyperparameterMetricTag: rmse
    goal: MINIMIZE
    maxTrials: 4 #20
    maxParallelTrials: 2 #5
    enableTrialEarlyStopping: True
    algorithm: RANDOM_SEARCH
    params:
      - parameterName: learning-rate
        type: DOUBLE
        minValue: 0.00001
        maxValue: 0.005
        scaleType: UNIT_LOG_SCALE
      - parameterName: first-layer-size
        type: INTEGER
        minValue: 50
        maxValue: 500
        scaleType: UNIT_LINEAR_SCALE
      - parameterName: num-layers
        type: INTEGER
        minValue: 1
        maxValue: 15
        scaleType: UNIT_LINEAR_SCALE
      - parameterName: scale-factor
        type: DOUBLE
        minValue: 0.1
        maxValue: 1.0
        scaleType: UNIT_REVERSE_LOG_SCALE
```

- The following code executes the training job on Cloud MLE.

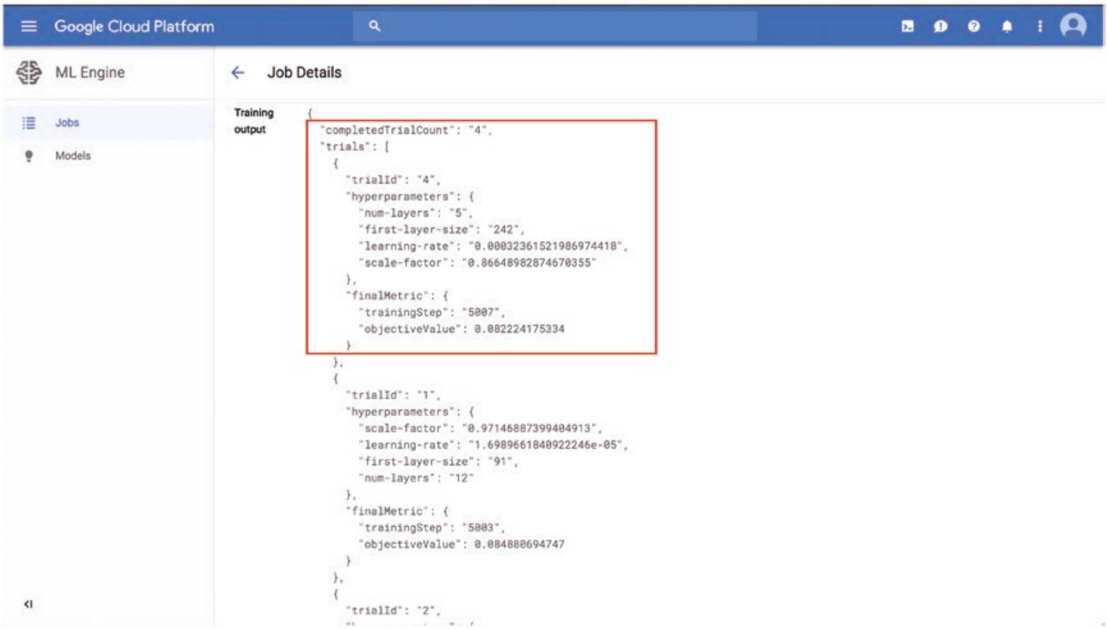
```
%%bash
JOB_NAME=superconductor_$(date -u +%y%m%d_%H%M%S)
HPTUNING_CONFIG=hptuning_config.yaml
GCS_JOB_DIR=gs://$bucket_name/jobs/$JOB_NAME

echo $GCS_JOB_DIR

gcloud ai-platform jobs submit training $JOB_NAME \
                                    --stream-logs \
                                    --runtime-version $tf_version \
                                    --job-dir $GCS_JOB_DIR \
                                    --module-name trainer.task \
                                    --package-path trainer/ \
                                    --region us-central1 \
                                    --scale-tier=STANDARD_1 \
                                    --config $HPTUNING_CONFIG \
                                    -- \
                                    --train-steps 5000 \
                                    --eval-steps 100

gs://superconductor/jobs/superconductor_181222_040429
endTime: '2018-12-22T04:24:50'
jobId: superconductor_181222_040429
startTime: '2018-12-22T04:04:35'
state: SUCCEEDED
```

- Cloud MLE training output is shown in Figure 44-5.



```
{
  "completedTrialCount": "4",
  "trials": [
    {
      "trialId": "4",
      "hyperparameters": {
        "num-layers": "5",
        "first-layer-size": "242",
        "learning-rate": "0.000323615219869744l8",
        "scale-factor": "0.86648982874670355"
      },
      "finalMetric": {
        "trainingStep": "5007",
        "objectiveValue": 0.082224175334
      }
    },
    {
      "trialId": "1",
      "hyperparameters": {
        "scale-factor": "0.97146887399404913",
        "learning-rate": "1.6989661840922246e-05",
        "first-layer-size": "91",
        "num-layers": "12"
      },
      "finalMetric": {
        "trainingStep": "5003",
        "objectiveValue": 0.084880694747
      }
    },
    {
      "trialId": "2",
```

Figure 44-5. *Cloud MLE training output*

Deploy Trained Model

The best model trial with the lowest **objectiveValue** is deployed for inference on Cloud MLE:

- Display content of selected trained model directory.

  ```
  %%bash
  gsutil ls gs://${BUCKET}/jobs/superconductor_181222_040429/4/
  export/superconductor/1545452450
  ```

  ```
  'Output':
  gs://superconductor/jobs/superconductor_181222_040429/4/export/
  superconductor/1545452450/
  gs://superconductor/jobs/superconductor_181222_040429/4/export/
  superconductor/1545452450/saved_model.pb
  gs://superconductor/jobs/superconductor_181222_040429/4/export/
  superconductor/1545452450/variables/
  ```