Predicting Housing Prices using Tensorflow + Cloud ML Engine

This notebook will show you how to create a tensorflow model, train it on the cloud in a distributed fashion across multiple CPUs or GPUs, explore the results using Tensorboard, and finally deploy the model for online prediction. We will demonstrate this by building a model to predict housing prices.

In [1]:

```
import pandas as pd
import tensorflow as tf
```

```
/usr/local/lib/python3.5/dist-packages/tensorflow/python/framework/dtypes.py:516: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.5/dist-packages/tensorflow/python/framework/dtypes.py:517: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.5/dist-packages/tensorflow/python/framework/dtypes.py:518: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.5/dist-packages/tensorflow/python/framework/dtypes.py:519: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.5/dist-packages/tensorflow/python/framework/dtypes.py:520: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.5/dist-packages/tensorflow/python/framework/dtypes.py:525: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
/usr/local/lib/python3.5/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:541: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.5/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:542: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.5/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:543: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.5/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:544: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.5/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:545: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.5/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:550: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

In [2]:

```
print(tf.__version__)
```

```
1.14.0
```

Tensorflow APIs

(image: https://www.tensorflow.org/images/tensorflow_programming_environment.png)

Tensorflow is a heirarchical framework. The further down the heirarchy you go, the more flexibility you have, but that more code you have to write. A best practice is start at the highest level of abstraction. Then if you need additional flexibility for some reason drop down one layer.

For this tutorial we will be operating at the highest level of Tensorflow abstraction, using the Estimator API.

Steps

1. Load raw data
2. Write Tensorflow Code
   A. Define Feature Columns
   B. Define Estimator
   C. Define Input Function
   D. Define Serving Function
   E. Define Train and Eval Function
3. Package Code
4. Train
5. Inspect Results
6. Deploy Model
7. Get Predictions

1) Load Raw Data

This is a publically available dataset on housing prices in Boston area suburbs circa 1978. It is hosted in a Google Cloud Storage bucket.

For datasets too large to fit in memory you would read the data in batches. Tensorflow provides a queueing mechanism for this which is documented [here (https://www.tensorflow.org/programmers_guide/reading_data)](https://www.tensorflow.org/programmers_guide/reading_data).

In our case the dataset is small enough to fit in memory so we will simply read it into a pandas dataframe.

In [3]:

```
#downlad data from GCS and store as pandas dataframe
data_train = pd.read_csv(
  filepath_or_buffer='https://storage.googleapis.com/vijay-public/boston_housing/housing_train.csv',
  names=["CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS","RAD","TAX","PTRATIO","MEDV"])

data_test = pd.read_csv(
  filepath_or_buffer='https://storage.googleapis.com/vijay-public/boston_housing/housing_test.csv',
  names=["CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS","RAD","TAX","PTRATIO","MEDV"])
```

In [4]:

```
data_train.head()
```

Out[4]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	24000
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	21600
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	34700
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	33400
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	36200

Column Descriptions:

1. CRIM: per capita crime rate by town
2. ZN: proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS: proportion of non-retail business acres per town
4. CHAS: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
5. NOX: nitric oxides concentration (parts per 10 million)
6. RM: average number of rooms per dwelling
7. AGE: proportion of owner-occupied units built prior to 1940
8. DIS: weighted distances to five Boston employment centres
9. RAD: index of accessibility to radial highways
10. TAX: full-value property-tax rate per $10,000
11. PTRATIO: pupil-teacher ratio by town
12. MEDV: Median value of owner-occupied homes

2) Write Tensorflow Code

2.A Define Feature Columns

Feature columns are your Estimator's data "interface." They tell the estimator in what format they should expect data and how to interpret it (is it one-hot? sparse? dense? continous?). [https://www.tensorflow.org/api_docs/python/tf/feature_column (https://www.tensorflow.org/api_docs/python/tf/feature_column)](https://www.tensorflow.org/api_docs/python/tf/feature_column)

In [5]:

```
FEATURES = ["CRIM", "ZN", "INDUS", "NOX", "RM",
            "AGE", "DIS", "TAX", "PTRATIO"]
LABEL = "MEDV"

feature_cols = [tf.feature_column.numeric_column(k)
                  for k in FEATURES] #list of Feature Columns
```

2.B Define Estimator

An Estimator is what actually implements your training, eval and prediction loops. Every estimator has the following methods:

- fit() for training
- eval() for evaluation
- predict() for prediction
- export_savedmodel() for writing model state to disk

Tensorflow has several canned estimator that already implement these methods (DNNClassifier, LogisticClassifier etc..) or you can implement a custom estimator. Instructions on how to implement a custom estimator [here (https://www.tensorflow.org/extend/estimators)](https://www.tensorflow.org/extend/estimators) and see an example [here (https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/blogs/timeseries/rnn_cloudmle.ipynb)](https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/blogs/timeseries/rnn_cloudmle.ipynb).

For simplicity we will use a canned estimator. To instantiate an estimator simply pass it what Feature Columns to expect and specify a directory for it to output to.

Notice we wrap the estimator with a function. This is to allow us to specify the 'output_dir' at runtime, instead of having to hardcode it here

In [6]:

```
def generate_estimator(output_dir):
  return tf.estimator.DNNRegressor(feature_columns=feature_cols,
                                            hidden_units=[10, 10],
                                            model_dir=output_dir)
```

2.C Define Input Function

Now that you have an estimator and it knows what type of data to expect and how to intepret, you need to actually pass the data to it! This is the job of the input function.

The input function returns a (features, label) tuple

- features: A python dictionary. Each key is a feature column name and its value is the tensor containing the data for that Feature
- label: A Tensor containing the label column

In [7]:

```
def generate_input_fn(data_set):
    def input_fn():
      features = {k: tf.constant(data_set[k].values) for k in FEATURES}
      labels = tf.constant(data_set[LABEL].values)
      return features, labels
    return input_fn
```

2.D Define Serving Input Function

To predict with the model, we need to define a serving input function which will be used to read inputs from a user at prediction time.

Why do we need a separate serving function? Don't we input the same features during training as in serving?

Yes, but we may be *receiving* data in a different format during serving. The serving input function preforms transormations neccessary to get the data provided at prediction time into the format compatible with the Estimator API.

returns a (features, inputs) tuple

- features: A dict of features to be passed to the Estimator
- inputs: A dictionary of inputs the predictions server should expect from the user

In [8]:

```
def serving_input_fn():
  #feature_placeholders are what the caller of the predict() method will have to provide
  feature_placeholders = {
      column.name: tf.placeholder(column.dtype, [None])
      for column in feature_cols
  }

  #features are what we actually pass to the estimator
  features = {
    # Inputs are rank 1 so that we can provide scalars to the server
    # but Estimator expects rank 2, so we expand dimension
    key: tf.expand_dims(tensor, -1)
    for key, tensor in feature_placeholders.items()
  }
  return tf.estimator.export.ServingInputReceiver(
    features, feature_placeholders
  )
```

2.E Define Train and Eval Function

Finally to train and evaluate we use tf.estimator.train_and_evaluate()

This function is special because it provides consistent behavior across local and distributed environments.

Meaning if you run on multiple CPUs or GPUs, it takes care of parrallelizing the computation graph across these devices for you!

The tran_and_evaluate() function requires three arguments:

- estimator: we already defined this earlier
- train_spec: specifies the training input function
- eval_spec: specifies the eval input function, and also an 'exporter' which uses our serving_input_fn for serving the model

Note running this cell will give an error because we haven't specified an output_dir, we will do that later

In [ ]:

3) Package Code

You've now written all the tensoflow code you need!

To make it compatible with Cloud ML Engine we'll combine the above tensorflow code into a single python file with two simple changes

1. Add some boilerplate code to parse the command line arguments required for gcloud.
2. Use the learn_runner.run() function to run the experiment

We also add an empty __init__.py file to the folder. This is just the python convention for identifying modules.

In [9]:

```
%%bash
mkdir trainer
touch trainer/__init__.py
```

In [10]:

```
%%writefile trainer/task.py

import argparse
import pandas as pd
import tensorflow as tf
from tensorflow.contrib.learn.python.learn import learn_runner
from tensorflow.contrib.learn.python.learn.utils import saved_model_export_utils

print(tf.__version__)
tf.logging.set_verbosity(tf.logging.ERROR)

data_train = pd.read_csv(
  filepath_or_buffer='https://storage.googleapis.com/vijay-public/boston_housing/housing_train.csv',
  names=["CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS","RAD","TAX","PTRATIO","MEDV"])

data_test = pd.read_csv(
  filepath_or_buffer='https://storage.googleapis.com/vijay-public/boston_housing/housing_test.csv',
  names=["CRIM","ZN","INDUS","CHAS","NOX","RM","AGE","DIS","RAD","TAX","PTRATIO","MEDV"])

FEATURES = ["CRIM", "ZN", "INDUS", "NOX", "RM",
            "AGE", "DIS", "TAX", "PTRATIO"]
LABEL = "MEDV"

feature_cols = [tf.feature_column.numeric_column(k)
                  for k in FEATURES] #list of Feature Columns

def generate_estimator(output_dir):
  return tf.estimator.DNNRegressor(feature_columns=feature_cols,
                                            hidden_units=[10, 10],
                                            model_dir=output_dir)

def generate_input_fn(data_set):
    def input_fn():
      features = {k: tf.constant(data_set[k].values) for k in FEATURES}
      labels = tf.constant(data_set[LABEL].values)
      return features, labels
    return input_fn

def serving_input_fn():
  #feature_placeholders are what the caller of the predict() method will have to provide
  feature_placeholders = {
      column.name: tf.placeholder(column.dtype, [None])
      for column in feature_cols
  }

  #features are what we actually pass to the estimator
  features = {
    # Inputs are rank 1 so that we can provide scalars to the server
    # but Estimator expects rank 2, so we expand dimension
    key: tf.expand_dims(tensor, -1)
    for key, tensor in feature_placeholders.items()
  }
  return tf.estimator.export.ServingInputReceiver(
    features, feature_placeholders
  )

train_spec = tf.estimator.TrainSpec(
                input_fn=generate_input_fn(data_train),
                max_steps=3000)

exporter = tf.estimator.LatestExporter('Servo', serving_input_fn)

eval_spec=tf.estimator.EvalSpec(
            input_fn=generate_input_fn(data_test),
            steps=1,
            exporters=exporter)

######START CLOUD ML ENGINE BOILERPLATE######
if __name__ == '__main__':
  parser = argparse.ArgumentParser()
  # Input Arguments
  parser.add_argument(
      '--output_dir',
      help='GCS location to write checkpoints and export models',
      required=True
  )
  parser.add_argument(
        '--job-dir',
        help='this model ignores this field, but it is required by gcloud',
        default='junk'
    )
  args = parser.parse_args()
  arguments = args.__dict__
  output_dir = arguments.pop('output_dir')
######END CLOUD ML ENGINE BOILERPLATE######

  #initiate training job
  tf.estimator.train_and_evaluate(generate_estimator(output_dir), train_spec, eval_spec)
```

```
Writing trainer/task.py
```

4) Train

Now that our code is packaged we can invoke it using the gcloud command line tool to run the training.

Note: Since our dataset is so small and our model is simple the overhead of provisioning the cluster is longer than the actual training time. Accordingly you'll notice the single VM cloud training takes longer than the local training, and the distributed cloud training takes longer than single VM cloud. For larger datasets and more complex models this will reverse

Set Environment Vars

We'll create environment variables for our project name GCS Bucket and reference this in future commands.

If you do not have a GCS bucket, you can create one using [these (https://cloud.google.com/storage/docs/creating-buckets)](https://cloud.google.com/storage/docs/creating-buckets) instructions.

In [11]:

```
GCS_BUCKET = 'gs://qwiklabs-gcp-01-8bce7d1ca614' #CHANGE THIS TO YOUR BUCKET
PROJECT = 'qwiklabs-gcp-01-8bce7d1ca614' #CHANGE THIS TO YOUR PROJECT ID
REGION = 'us-central1' #OPTIONALLY CHANGE THIS
```

In [12]:

```
import os
os.environ['GCS_BUCKET'] = GCS_BUCKET
os.environ['PROJECT'] = PROJECT
os.environ['REGION'] = REGION
```

Run local

It's a best practice to first run locally on a small dataset to check for errors. Note you can ignore the warnings in this case, as long as there are no errors.

In [13]:

```
%%bash
gcloud ai-platform local train \
   --module-name=trainer.task \
   --package-path=trainer \
   -- \
   --output_dir='./output'
```

```
1.14.0

WARNING:tensorflow:From /home/jupyter/tensorflow_teaching_examples/housing_prices/trainer/task.py:9: The name tf.logging.set_verbosity is deprecated. Please use tf.compat.v1.logging.set_verbosity instead.

WARNING:tensorflow:From /home/jupyter/tensorflow_teaching_examples/housing_prices/trainer/task.py:9: The name tf.logging.ERROR is deprecated. Please use tf.compat.v1.logging.ERROR instead.

2020-03-30 10:21:57.425304: I tensorflow/core/platform/cpu_feature_guard.cc:145] This TensorFlow binary is optimized with Intel(R) MKL-DNN to use the following CPU instructions in performance critical operations:  AVX2 FMA
To enable them in non-MKL-DNN operations, rebuild TensorFlow with the appropriate compiler flags.

User settings:

   KMP_AFFINITY=granularity=fine,verbose,compact,1,0
   KMP_BLOCKTIME=0
   KMP_SETTINGS=1
   OMP_NUM_THREADS=4

Effective settings:

   KMP_ABORT_DELAY=0
   KMP_ADAPTIVE_LOCK_PROPS='1,1024'
   KMP_ALIGN_ALLOC=64
   KMP_ALL_THREADPRIVATE=128
   KMP_ATOMIC_MODE=2
   KMP_BLOCKTIME=0
   KMP_CPUINFO_FILE: value is not defined
   KMP_DETERMINISTIC_REDUCTION=false
   KMP_DEVICE_THREAD_LIMIT=2147483647
   KMP_DISP_HAND_THREAD=false
   KMP_DISP_NUM_BUFFERS=7
   KMP_DUPLICATE_LIB_OK=false
   KMP_FORCE_REDUCTION: value is not defined
   KMP_FOREIGN_THREADS_THREADPRIVATE=true
   KMP_FORKJOIN_BARRIER='2,2'
   KMP_FORKJOIN_BARRIER_PATTERN='hyper,hyper'
   KMP_FORKJOIN_FRAMES=true
   KMP_FORKJOIN_FRAMES_MODE=3
   KMP_GTID_MODE=3
   KMP_HANDLE_SIGNALS=false
   KMP_HOT_TEAMS_MAX_LEVEL=1
   KMP_HOT_TEAMS_MODE=0
   KMP_INIT_AT_FORK=true
   KMP_INIT_WAIT=2048
   KMP_ITT_PREPARE_DELAY=0
   KMP_LIBRARY=throughput
   KMP_LOCK_KIND=queuing
   KMP_MALLOC_POOL_INCR=1M
   KMP_NEXT_WAIT=1024
   KMP_NUM_LOCKS_IN_BLOCK=1
   KMP_PLAIN_BARRIER='2,2'
   KMP_PLAIN_BARRIER_PATTERN='hyper,hyper'
   KMP_REDUCTION_BARRIER='1,1'
   KMP_REDUCTION_BARRIER_PATTERN='hyper,hyper'
   KMP_SCHEDULE='static,balanced;guided,iterative'
   KMP_SETTINGS=true
   KMP_SPIN_BACKOFF_PARAMS='4096,100'
   KMP_STACKOFFSET=64
   KMP_STACKPAD=0
   KMP_STACKSIZE=4M
   KMP_STORAGE_MAP=false
   KMP_TASKING=2
   KMP_TASKLOOP_MIN_TASKS=0
   KMP_TASK_STEALING_CONSTRAINT=1
   KMP_TEAMS_THREAD_LIMIT=4
   KMP_TOPOLOGY_METHOD=all
   KMP_USER_LEVEL_MWAIT=false
   KMP_VERSION=false
   KMP_WARNINGS=true
   OMP_AFFINITY_FORMAT='OMP: pid %P tid %T thread %n bound to OS proc set {%a}'
   OMP_ALLOCATOR=omp_default_mem_alloc
   OMP_CANCELLATION=false
   OMP_DEFAULT_DEVICE=0
   OMP_DISPLAY_AFFINITY=false
   OMP_DISPLAY_ENV=false
   OMP_DYNAMIC=false
   OMP_MAX_ACTIVE_LEVELS=2147483647
   OMP_MAX_TASK_PRIORITY=0
   OMP_NESTED=false
   OMP_NUM_THREADS='4'
   OMP_PLACES: value is not defined
   OMP_PROC_BIND='intel'
   OMP_SCHEDULE='static'
   OMP_STACKSIZE=4M
   OMP_TARGET_OFFLOAD=DEFAULT
   OMP_THREAD_LIMIT=2147483647
   OMP_TOOL=enabled
   OMP_TOOL_LIBRARIES: value is not defined
   OMP_WAIT_POLICY=PASSIVE
   KMP_AFFINITY='verbose,warnings,respect,granularity=fine,compact,1,0'

2020-03-30 10:21:57.469232: I tensorflow/core/platform/profile_utils/cpu_utils.cc:94] CPU Frequency: 2200000000 Hz
2020-03-30 10:21:57.469547: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x55910f24ab70 executing computations on platform Host. Devices:
2020-03-30 10:21:57.469577: I tensorflow/compiler/xla/service/service.cc:175]   StreamExecutor device (0): <undefined>, <undefined>
OMP: Info #212: KMP_AFFINITY: decoding x2APIC ids.
OMP: Info #210: KMP_AFFINITY: Affinity capable, using global cpuid leaf 11 info
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: 0-3
OMP: Info #156: KMP_AFFINITY: 4 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 1 packages x 2 cores/pkg x 2 threads/core (2 total cores)
OMP: Info #214: KMP_AFFINITY: OS proc to physical thread map:
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 1 thread 1
OMP: Info #250: KMP_AFFINITY: pid 1998 tid 1998 thread 0 bound to OS proc set 0
2020-03-30 10:21:57.470879: I tensorflow/core/common_runtime/process_util.cc:115] Creating new thread pool with default inter op setting: 2. Tune using inter_op_parallelism_threads for best performance.
2020-03-30 10:21:57.558864: W tensorflow/compiler/jit/mark_for_compilation_pass.cc:1412] (One-time warning): Not using XLA:CPU for cluster because envvar TF_XLA_FLAGS=--tf_xla_cpu_global_jit was not set.  If you want XLA:CPU, either set that envvar, or use experimental_jit_scope to enable XLA:CPU.  To confirm that XLA is active, pass --v
module=xla_compilation_cache=1 (as a proper command-line flag, not via TF_XLA_FLAGS) or set the envvar XLA_FLAGS=--xla_hlo_profile.
OMP: Info #250: KMP_AFFINITY: pid 1998 tid 2028 thread 1 bound to OS proc set 1
OMP: Info #250: KMP_AFFINITY: pid 1998 tid 2070 thread 3 bound to OS proc set 3
OMP: Info #250: KMP_AFFINITY: pid 1998 tid 2069 thread 2 bound to OS proc set 2
OMP: Info #250: KMP_AFFINITY: pid 1998 tid 2071 thread 4 bound to OS proc set 0
OMP: Info #250: KMP_AFFINITY: pid 1998 tid 2029 thread 5 bound to OS proc set 1
OMP: Info #250: KMP_AFFINITY: pid 1998 tid 2072 thread 6 bound to OS proc set 2
OMP: Info #250: KMP_AFFINITY: pid 1998 tid 2073 thread 7 bound to OS proc set 3
OMP: Info #250: KMP_AFFINITY: pid 1998 tid 2074 thread 8 bound to OS proc set 0
```

Run on cloud (1 cloud ML unit)

First we specify which GCP project to use.

In [14]:

```
%%bash
gcloud config set project $PROJECT
```

```
Updated property [core/project].
```

Then we specify which GCS bucket to write to and a job name. Job names submitted to the ml engine must be project unique, so we append the system date/time. Update the cell below to point to a GCS bucket you own.

In [15]:

```
%%bash
JOBNAME=housing_$(date -u +%y%m%d_%H%M%S)

gcloud ai-platform jobs submit training $JOBNAME \
   --region=$REGION \
   --module-name=trainer.task \
   --package-path=./trainer \
   --job-dir=$GCS_BUCKET/$JOBNAME/ \
   --runtime-version 1.15 \
   -- \
   --output_dir=$GCS_BUCKET/$JOBNAME/output
```

```
jobId: housing_200330_102205
state: QUEUED

Job [housing_200330_102205] submitted successfully.
Your job is still active. You may view the status of your job with the command

  $ gcloud ai-platform jobs describe housing_200330_102205

or continue streaming the logs with the command

  $ gcloud ai-platform jobs stream-logs housing_200330_102205
```

Run on cloud (10 cloud ML units)

Because we are using the TF Estimators interface, distributed computing just works! The only change we need to make to run in a distributed fashion is to add the [--scale-tier (https://cloud.google.com/ml/pricing#ml_training_units_by_scale_tier)](https://cloud.google.com/ml/pricing#ml_training_units_by_scale_tier) argument. Cloud ML Engine then takes care of distributing the training across devices for you!

In [16]:

```
%%bash
JOBNAME=housing_$(date -u +%y%m%d_%H%M%S)

gcloud ai-platform jobs submit training $JOBNAME \
   --region=$REGION \
   --module-name=trainer.task \
   --package-path=./trainer \
   --job-dir=$GCS_BUCKET/$JOBNAME \
   --runtime-version 1.15 \
   --scale-tier=STANDARD_1 \
   -- \
   --output_dir=$GCS_BUCKET/$JOBNAME/output
```

```
jobId: housing_200330_102207
state: QUEUED

Job [housing_200330_102207] submitted successfully.
Your job is still active. You may view the status of your job with the command

  $ gcloud ai-platform jobs describe housing_200330_102207

or continue streaming the logs with the command

  $ gcloud ai-platform jobs stream-logs housing_200330_102207
```

Run on cloud GPU (3 cloud ML units)

Also works with GPUs!

"BASIC_GPU" corresponds to one Tesla K80 at the time of this writing, hardware subject to change. 1 GPU is charged as 3 cloud ML units.

In [17]:

```
%%bash
JOBNAME=housing_$(date -u +%y%m%d_%H%M%S)

gcloud ai-platform jobs submit training $JOBNAME \
   --region=$REGION \
   --module-name=trainer.task \
   --package-path=./trainer \
   --job-dir=$GCS_BUCKET/$JOBNAME \
   --runtime-version 1.15 \
   --scale-tier=BASIC_GPU \
   -- \
   --output_dir=$GCS_BUCKET/$JOBNAME/output
```

```
jobId: housing_200330_102209
state: QUEUED

Job [housing_200330_102209] submitted successfully.
Your job is still active. You may view the status of your job with the command

  $ gcloud ai-platform jobs describe housing_200330_102209

or continue streaming the logs with the command

  $ gcloud ai-platform jobs stream-logs housing_200330_102209
```

Run on 8 cloud GPUs (24 cloud ML units)

To train across multiple GPUs you use a [custom scale tier (https://cloud.google.com/ml/docs/concepts/training-overview#job_configuration_parameters)](https://cloud.google.com/ml/docs/concepts/training-overview#job_configuration_parameters).

You specify the number and types of machines you want to run on in a config.yaml, then reference that config.yaml via the --config config.yaml command line argument.

Here I am specifying a master node with machine type complex_model_m_gpu and one worker node of the same type. Each complex_model_m_gpu has 4 GPUs so this job will run on 2x4=8 GPUs total.

WARNING: The default project quota is 10 cloud ML units, so unless you have requested a quota increase you will get a quota exceeded error. This command is just for illustrative purposes.

In [18]:

```
%%writefile config.yaml
trainingInput:
  scaleTier: CUSTOM
  masterType: complex_model_m_gpu
  workerType: complex_model_m_gpu
  workerCount: 1
```

```
Writing config.yaml
```

In [19]:

```
%%bash
JOBNAME=housing_$(date -u +%y%m%d_%H%M%S)

gcloud ai-platform jobs submit training $JOBNAME \
   --region=$REGION \
   --module-name=trainer.task \
   --package-path=./trainer \
   --job-dir=$GCS_BUCKET/$JOBNAME \
   --runtime-version 1.15 \
   --config config.yaml \
   -- \
   --output_dir=$GCS_BUCKET/$JOBNAME/output
```

```
jobId: housing_200330_102211
state: QUEUED

Job [housing_200330_102211] submitted successfully.
Your job is still active. You may view the status of your job with the command

  $ gcloud ai-platform jobs describe housing_200330_102211

or continue streaming the logs with the command

  $ gcloud ai-platform jobs stream-logs housing_200330_102211
```

5) Inspect Results Using Tensorboard

Tensorboard is a utility that allows you to visualize your results.

Expand the 'loss' graph. What is your evaluation loss? This is squared error, so take the square root of it to get the average error in dollars. Does this seem like a reasonable margin of error for predicting a housing price?

To activate TensorBoard within the JupyterLab UI navigate to **File** - **New Launcher**. Then double-click the 'Tensorboard' icon on the bottom row.

TensorBoard 1 will appear in the new tab. Navigate through the three tabs to see the active TensorBoard. The 'Graphs' and 'Projector' tabs offer very interesting information including the ability to replay the tests.

You may close the TensorBoard tab when you are finished exploring.

6) Deploy Model For Predictions

Cloud ML Engine has a prediction service that will wrap our tensorflow model with a REST API and allow remote clients to get predictions.

You can deploy the model from the Google Cloud Console GUI, or you can use the gcloud command line tool. We will use the latter method. Note this will take up to 5 minutes.

In [20]:

```
%%bash
MODEL_NAME="housing_prices"
MODEL_VERSION="v1"
MODEL_LOCATION=output/export/Servo/$(ls output/export/Servo | tail -1)

#gcloud ai-platform versions delete ${MODEL_VERSION} --model ${MODEL_NAME} #Uncomment to overwrite existing version
#gcloud ai-platform models delete ${MODEL_NAME} #Uncomment to overwrite existing model
gcloud ai-platform models create ${MODEL_NAME} --regions $REGION
gcloud ai-platform versions create ${MODEL_VERSION} --model ${MODEL_NAME} --origin ${MODEL_LOCATION} --staging-bucket=$GCS_BUCKET
```

```
Created ml engine model [projects/qwiklabs-gcp-01-8bce7d1ca614/models/housing_prices].
Creating version (this might take a few minutes)......
........................................................................................................................................................................................................................................
done.
```

7) Get Predictions

There are two flavors of the ML Engine Prediction Service: Batch and online.

Online prediction is more appropriate for latency sensitive requests as results are returned quickly and synchronously.

Batch prediction is more appropriate for large prediction requests that you only need to run a few times a day.

The prediction services expects prediction requests in standard JSON format so first we will create a JSON file with a couple of housing records.

In [21]:

```
%%writefile records.json
{"CRIM": 0.00632,"ZN": 18.0,"INDUS": 2.31,"NOX": 0.538, "RM": 6.575, "AGE": 65.2, "DIS": 4.0900, "TAX": 296.0, "PTRATIO": 15.3}
{"CRIM": 0.00332,"ZN": 0.0,"INDUS": 2.31,"NOX": 0.437, "RM": 7.7, "AGE": 40.0, "DIS": 5.0900, "TAX": 250.0, "PTRATIO": 17.3}
```

```
Writing records.json
```

Now we will pass this file to the prediction service using the gcloud command line tool. Results are returned immediatley!

In [22]:

```
!gcloud ai-platform predict --model housing_prices --json-instances records.json
```

```
PREDICTIONS
[27386.294921875]
[21106.494140625]
```

Conclusion

What we covered

1. How to use Tensorflow's high level Estimator API
2. How to deploy tensorflow code for distributed training in the cloud
3. How to evaluate results using TensorBoard
4. How deploy the resulting model to the cloud for online prediction

What we didn't cover

1. How to leverage larger than memory datasets using Tensorflow's queueing system
2. How to create synthetic features from our raw data to aid learning (Feature Engineering)
3. How to improve model performance by finding the ideal hyperparameters using Cloud ML Engine's HyperTune (https://cloud.google.com/ml-engine/docs/how-tos/using-hyperparameter-tuning) feature

This lab is a great start, but adding in the above concepts is critical in getting your models to production ready quality. These concepts are covered in Google's 1-week on-demand Tensorflow + Cloud ML course: https://www.coursera.org/learn/serverless-machine-learning-gcp (https://www.coursera.org/learn/serverless-machine-learning-gcp)