Google Cloud

# Estimator API
# Introduction to TensorFlow

Martin Gorner

# Learn how to...

# Learn how to...

Create production-ready machine learning models the easy way

# Learn how to...

Create production-ready machine learning models the easy way

Train on large datasets that do not fit in memory

# Learn how to...

Create production-ready machine learning models the easy way

Train on large datasets that do not fit in memory

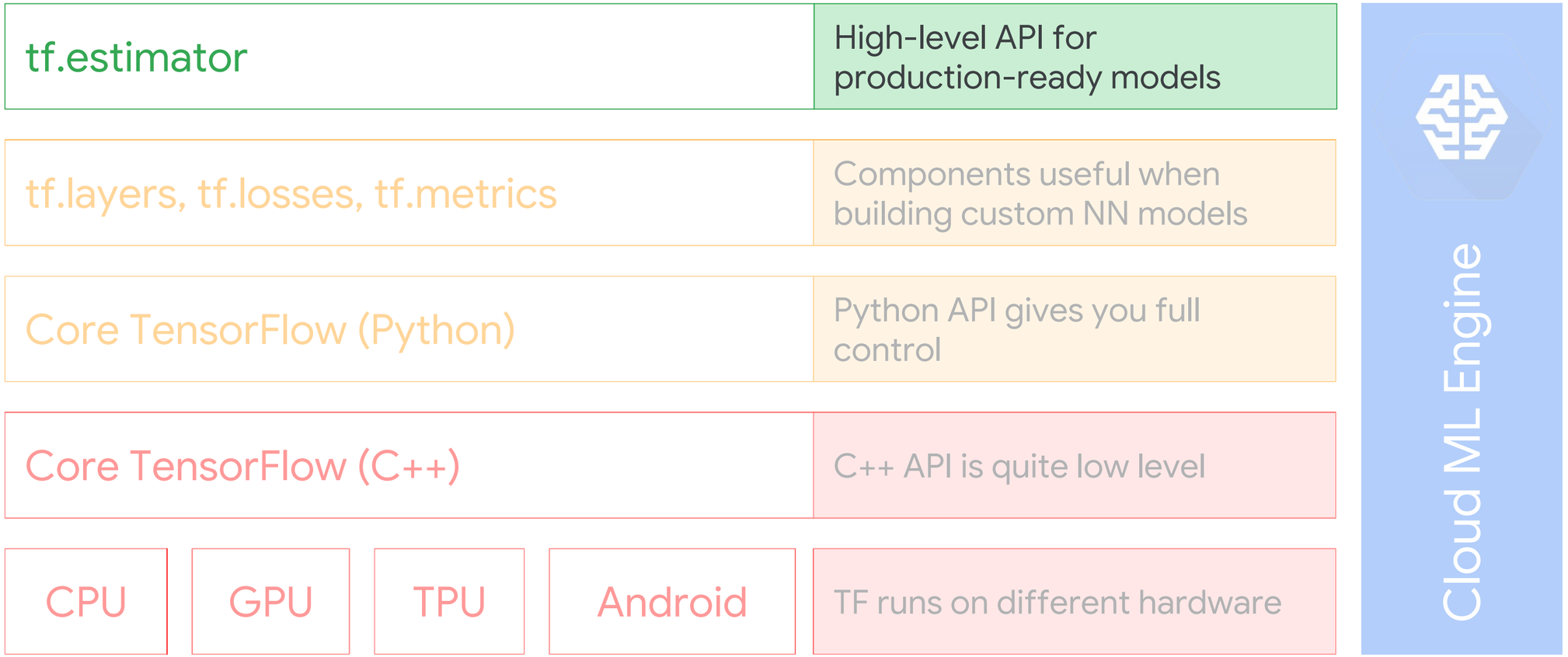
Monitor your training metrics in Tensorboard

Google Cloud

# Estimator API

Martin Gorner

# Estimators wrap up a large amount of boilerplate code, on top of the model itself



# From small to big to prod with the Estimator API

# From small to big to prod with the Estimator API

Quick model

# From small to big to prod with the Estimator API

Quick model

Checkpointing

# From small to big to prod with the Estimator API

Quick model

Checkpointing

Out-of-memory datasets

# From small to big to prod with the Estimator API

- Quick model
- Checkpointing
- Out-of-memory datasets
- Train / eval / monitor

# From small to big to prod with the Estimator API

- Quick model
- Checkpointing
- Out-of-memory datasets
- Train / eval / monitor
- Distributed training

# From small to big to prod with the Estimator API

- Quick model
- Checkpointing
- Out-of-memory datasets
- Train / eval / monitor
- Distributed training
- Hyper-parameter tuning on ML-Engine
- Production: serving predictions from a trained model

Pre-made estimators that can all be used in the same way

**tf.estimator.Estimator.**

Pre-made estimators that can all be used in the same way

**tf.estimator.Estimator.**

# Pre-made estimators that can all be used in the same way

**tf.estimator.Estimator.**

Pre-made regressors

```
LinearRegressor
DNNRegressor
DNNLinearCombinedRegressor
...
```

Pre-made classifiers

```
LinearClassifier
DNNClassifier
DNNLinearCombinedClassifier
```

your custom Estimator

```
...
```

# Pre-made estimators that can all be used in the same way

**tf.estimator.Estimator.**

Pre-made regressors

- `LinearRegressor`
- `DNNRegressor`
- `DNNLinearCombinedRegressor`
- `...`

Pre-made classifiers

- `LinearClassifier`
- `DNNClassifier`
- `DNNLinearCombinedClassifier`

your custom Estimator

- `...`

# Predict property value from historical data

# Predict property value from historical data

| “Features” | ??? |
| --- | --- |
| | ??? |

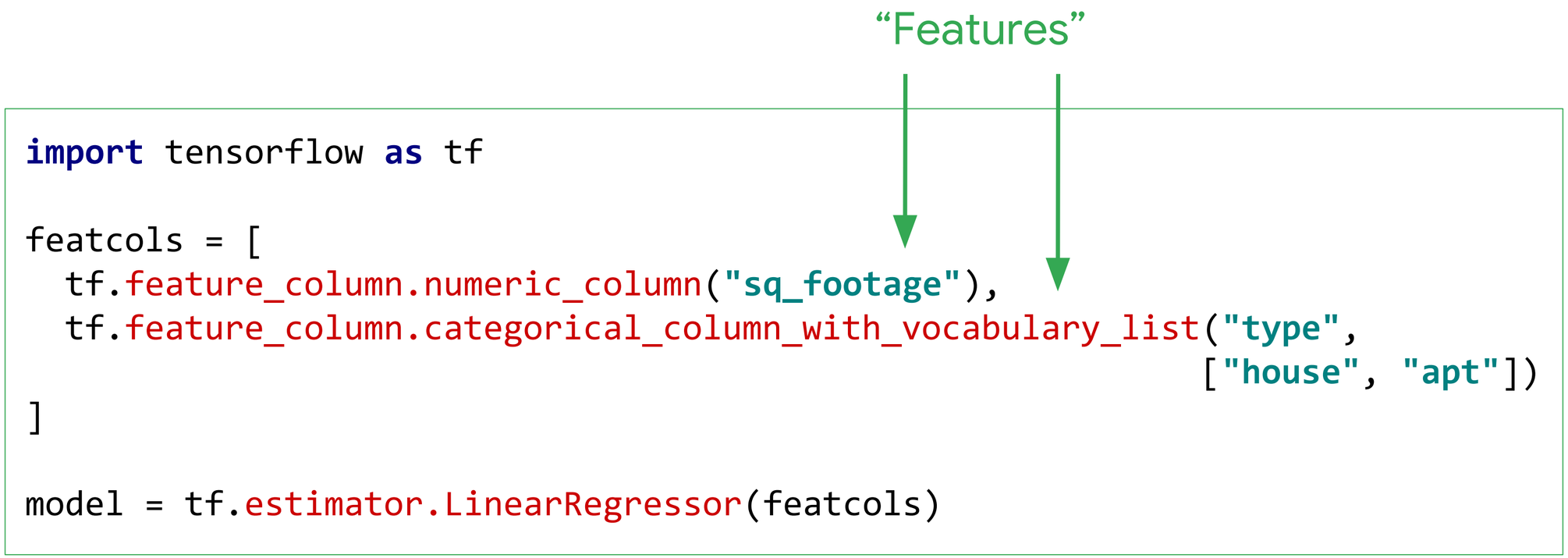
# Predict property value from historical data

| | |
|---|---|
| “Features” | Square footage<br>House / apartment |

**ML Model**

Output **Price:** $400,000

# Feature columns tell the model what inputs to expect



# Feature columns tell the model what inputs to expect

```
import tensorflow as tf

featcols = [
  tf.feature_column.numeric_column("sq_footage"),
  tf.feature_column.categorical_column_with_vocabulary_list("type",
                                                            ["house", "apt"])
]

model = tf.estimator.LinearRegressor(featcols)
```

# Feature columns tell the model what inputs to expect

```
import tensorflow as tf

featcols = [
  tf.feature_column.numeric_column("sq_footage"),
  tf.feature_column.categorical_column_with_vocabulary_list("type",
                                                            ["house", "apt"])
]

model = tf.estimator.LinearRegressor(featcols)
```

# Feature columns tell the model what inputs to expect

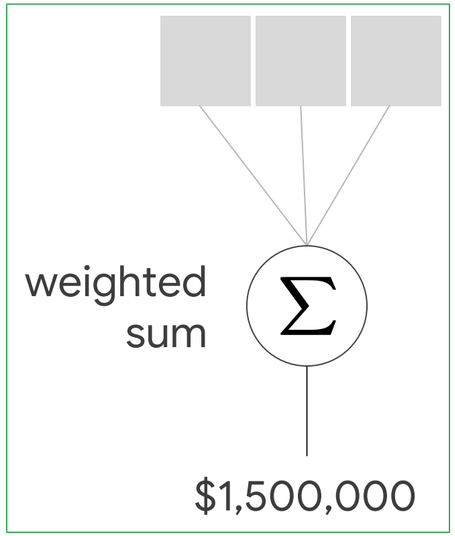
```
import tensorflow as tf

featcols = [
  tf.feature_column.numeric_column("sq_footage"),
  tf.feature_column.categorical_column_with_vocabulary_list("type",
                                                            ["house", "apt"])
]

model = tf.estimator.LinearRegressor(featcols)
```

# Feature columns tell the model what inputs to expect

```
import tensorflow as tf

featcols = [
  tf.feature_column.numeric_column("sq_footage"),
  tf.feature_column.categorical_column_with_vocabulary_list("type",
                                                            ["house", "apt"])
]

model = tf.estimator.LinearRegressor(featcols)
```

“Model”: predicts PRICE

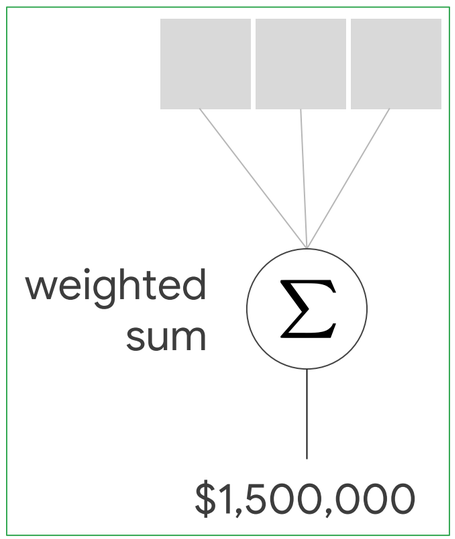
# Under the hood: feature columns take care of packing the inputs into the input vector of the model




`tf.estimator.LinearRegressor`

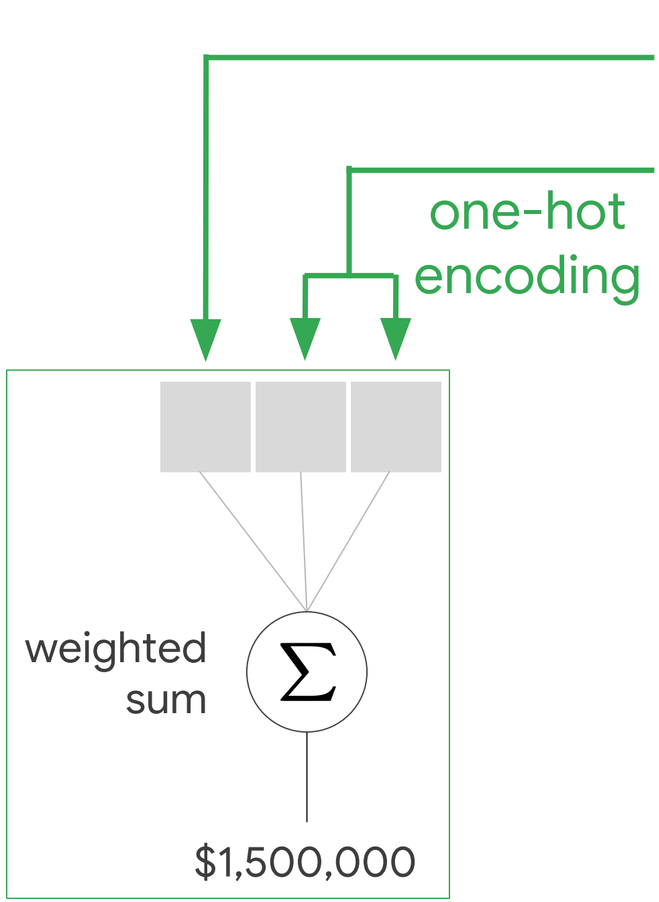
# Under the hood: feature columns take care of packing the inputs into the input vector of the model

```
tf.feature_column.numeric_column("sq_footage")
```




tf.estimator.LinearRegressor

# Under the hood: feature columns take care of packing the inputs into the input vector of the model



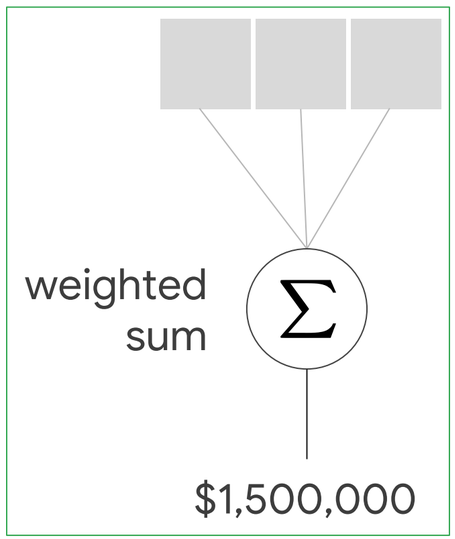

```
tf.feature_column.numeric_column("sq_footage")

tf.feature_column.categorical_column_with_vocabulary_list(
    "type", ["house", "apt"])
```

“ house” -> 1, 0

“ apt” -> 0, 1

tf.estimator.LinearRegressor

# Under the hood: feature columns take care of packing the inputs into the input vector of the model




```
tf.feature_column.numeric_column("sq_footage")

tf.feature_column.categorical_column_with_vocabulary_list(
    "type", ["house", "apt"])
```

one-hot encoding

“ house” -> 1, 0

“ apt” -> 0, 1

`tf.estimator.LinearRegressor`

Many more...

```
tf.feature_column.bucketized_column
tf.feature_column.embedding_column
Tf.feature_column.crossed_column
Tf.feature_column.categorical_column
_with_hash_bucket
...
```

# Training: feed in training input data and train for 100 epochs

```
def train_input_fn():
   features = {"sq_footage": [ 1000,    2000,    3000,    1000,  2000,  3000],
               "type":       ["house", "house", "house", "apt", "apt", "apt"]}
                               # prices in thousands
   labels =                  [ 500,     1000,    1500,    700,   1300,  1900]
   return features, labels



model.train(train_input_fn, steps=100)
```

# Training: feed in training input data and train for 100 epochs

```
def train_input_fn():
   features = {"sq_footage": [ 1000,    2000,    3000,    1000,  2000,  3000],
               "type":       ["house", "house", "house", "apt", "apt", "apt"]}
                               # prices in thousands
   labels =                  [ 500,     1000,    1500,    700,   1300,  1900]
   return features, labels



model.train(train_input_fn, steps=100)
```

# Predictions: once trained, the model can be used for prediction

```
def predict_input_fn():
   features = {"sq_footage": [1500,    1800],
               "type":       ["house", "apt"]}
   return features


predictions = model.predict(predict_input_fn)
```

# Predictions: once trained, the model can be used for prediction

```
def predict_input_fn():
   features = {"sq_footage": [1500,    1800],
               "type":        ["house", "apt"]}
   return features


predictions = model.predict(predict_input_fn)
```

# Predictions: once trained, the model can be used for prediction

```
def predict_input_fn():
   features = {"sq_footage": [1500,    1800],
               "type":       ["house", "apt"]}
   return features


predictions = model.predict(predict_input_fn)
```

generator

```
print(next(predictions))
print(next(predictions))
```

```
{'predictions': array([855.93], dtype=float32)}
{'predictions': array([859.07], dtype=float32)}
```

# Pick an Estimator, train, predict

```
import tensorflow as tf

featcols = [
  tf.feature_column.numeric_column("sq_footage"),
  tf.feature_column.categorical_column_with_vocabulary_list("type", ["house", "apt"])
]
```

# Pick an Estimator, train, predict

```
import tensorflow as tf

featcols = [
  tf.feature_column.numeric_column("sq_footage"),
  tf.feature_column.categorical_column_with_vocabulary_list("type", ["house", "apt"])
]

model = tf.estimator.LinearRegressor(featcols)
```

# Pick an Estimator, train, predict

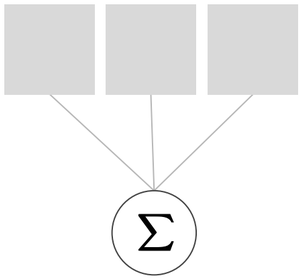
```
import tensorflow as tf

featcols = [
  tf.feature_column.numeric_column("sq_footage"),
  tf.feature_column.categorical_column_with_vocabulary_list("type", ["house", "apt"])
]

model = tf.estimator.LinearRegressor(featcols)

model.train(train_input_fn, steps=100)
```

# Pick an Estimator, train, predict

```
import tensorflow as tf

featcols = [
  tf.feature_column.numeric_column("sq_footage"),
  tf.feature_column.categorical_column_with_vocabulary_list("type", ["house", "apt"])
]

model = tf.estimator.LinearRegressor(featcols)

model.train(train_input_fn, steps=100)

predictions = model.predict(predict_input_fn)
```

To use a different pre-made estimator, just change the class name and supply appropriate parameters

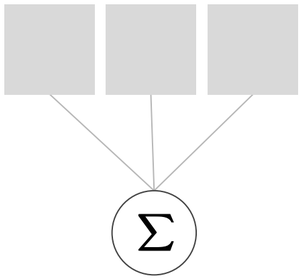
```
model = tf.estimator.DNNRegressor(featcols,
                                  hidden_units=[3, 2])
```




tf.estimator.
LinearRegressor

To use a different pre-made estimator, just change the class name and supply appropriate parameters

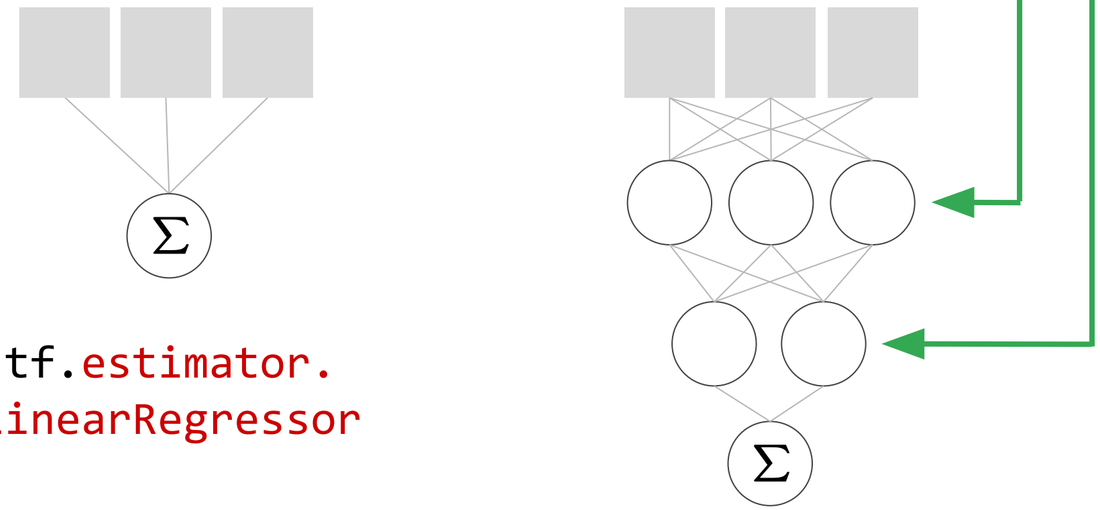
```
model = tf.estimator.DNNRegressor(featcols,
                                  hidden_units=[3, 2])
```




tf.estimator.
LinearRegressor

To use a different pre-made estimator, just change the class name and supply appropriate parameters

```
model = tf.estimator.DNNRegressor(featcols,
                                  hidden_units=[3, 2])
```




tf.estimator.
LinearRegressor

# For example, here are some of the things you can change about the DNN Regressor

```
tf.estimator.DNNRegressor(feature_columns=...,
                          hidden_units=[10, 5],
```

# For example, here are some of the things you can change about the DNN Regressor

```
tf.estimator.DNNRegressor(feature_columns=...,
                          hidden_units=[10, 5],
                          activation_fn=tf.nn.relu,
```

# For example, here are some of the things you can change about the DNN Regressor

```
tf.estimator.DNNRegressor(feature_columns=...,
                          hidden_units=[10, 5],
                          activation_fn=tf.nn.relu,
                          dropout=0.2,
                          optimizer="Adam")
```

# Model checkpoints

1 Continue training

# Model checkpoints

1 Continue training

2 Resume on failure

# Model checkpoints

1. Continue training
2. Resume on failure
3. Predict from trained model

# Estimators automatically checkpoint training

Where to put the checkpoints

```
model = tf.estimator.LinearRegressor(featcols, './model_trained')

model.train(train_input_fn, steps=100)
```

```
%ls model_trained

checkpoint                                      model.ckpt-100.meta
graph.pbtxt                                     model.ckpt-1.data-00000-of-00001
model.ckpt-100.data-00000-of-00001              model.ckpt-1.index
model.ckpt-100.index                            model.ckpt-1.meta
```

# We can now restore and predict with the model

```
trained_model = tf.estimator.LinearRegressor(featcols, './model_trained')

predictions = trained_model.predict(pred_input_fn)
```

```
INFO:tensorflow:Restoring parameters from
model_trained/model.ckpt-100
```

```
{'predictions': array([855.93], dtype=float32)}
{'predictions': array([859.07], dtype=float32)}
```

# Training also resumes from the last checkpoint

```
model = tf.estimator.LinearRegressor(featcols, './model_trained')

model.train(train_input_fn, steps=100)
```

Training continues

# Training also resumes from the last checkpoint

```
model = tf.estimator.LinearRegressor(featcols, './model_trained')

model.train(train_input_fn, steps=100)
```



Training continues

```
INFO:tensorflow:Restoring parameters from
model_trained/model.ckpt-100
```

# In memory data: usually numpy arrays or Pandas dataframes - you can use them directly

```
def numpy_train_input_fn(sqft, prop_type, price):  #np arrays
   return tf.estimator.inputs.numpy_input_fn(
       x = {"sq_footage": sqft, "type": prop_type},
       y = price,
       batch_size=128,
       num_epochs=10,
       shuffle=True,
       queue_capacity=1000
   )
```

```
def pandas_train_input_fn(df):  # a Pandas dataframe
   return tf.estimator.inputs.pandas_input_fn(
       x = df, # “sq_footage”, “type” selected
               # automatically because of feature
               # columns definition
       y = df['price'],
       batch_size=128,
       num_epochs=10,
       shuffle=True,
       queue_capacity=1000
   )
```

# In memory data: usually numpy arrays or Pandas dataframes - you can use them directly

```
def numpy_train_input_fn(sqft, prop_type, price):  #np arrays
   return tf.estimator.inputs.numpy_input_fn(
       x = {"sq_footage": sqft, "type": prop_type},
       y = price,
       batch_size=128,
       num_epochs=10,
       shuffle=True,
       queue_capacity=1000
   )
```

```
def pandas_train_input_fn(df):  # a Pandas dataframe
   return tf.estimator.inputs.pandas_input_fn(
       x = df, # “sq_footage”, “type” selected
               # automatically because of feature
               # columns definition
       y = df['price'],
       batch_size=128,
       num_epochs=10,
       shuffle=True,
       queue_capacity=1000
   )
```

# In memory data: usually numpy arrays or Pandas dataframes - you can use them directly

```
def numpy_train_input_fn(sqft, prop_type, price):  #np arrays
   return tf.estimator.inputs.numpy_input_fn(
       x = {"sq_footage": sqft, "type": prop_type},
       y = price,
       batch_size=128,
       num_epochs=10,
       shuffle=True,
       queue_capacity=1000
   )
```

```
def pandas_train_input_fn(df):  # a Pandas dataframe
   return tf.estimator.inputs.pandas_input_fn(
       x = df, # “sq_footage”, “type” selected
               # automatically because of feature
               # columns definition
       y = df['price'],
       batch_size=128,
       num_epochs=10,
       shuffle=True,
       queue_capacity=1000
   )
```

# In memory data: usually numpy arrays or Pandas dataframes - you can use them directly

```
def numpy_train_input_fn(sqft, prop_type, price):  #np arrays
   return tf.estimator.inputs.numpy_input_fn(
       x = {"sq_footage": sqft, "type": prop_type},
       y = price,
       batch_size=128,
       num_epochs=10,
       shuffle=True,
       queue_capacity=1000
   )
```

```
def pandas_train_input_fn(df):  # a Pandas dataframe
   return tf.estimator.inputs.pandas_input_fn(
       x = df, # “sq_footage”, “type” selected
               # automatically because of feature
               # columns definition
       y = df['price'],
       batch_size=128,
       num_epochs=10,
       shuffle=True,
       queue_capacity=1000
   )
```

# In memory data: usually numpy arrays or Pandas dataframes - you can use them directly

```
def numpy_train_input_fn(sqft, prop_type, price):  #np arrays
   return tf.estimator.inputs.numpy_input_fn(
       x = {"sq_footage": sqft, "type": prop_type},
       y = price,
       batch_size=128,
       num_epochs=10,
       shuffle=True,
       queue_capacity=1000
   )
```

```
def pandas_train_input_fn(df):  # a Pandas dataframe
   return tf.estimator.inputs.pandas_input_fn(
       x = df, # “sq_footage”, “type” selected
               # automatically because of feature
               # columns definition
       y = df['price'],
       batch_size=128,
       num_epochs=10,
       shuffle=True,
       queue_capacity=1000
   )
```

# Training happens until input is exhausted or number of steps is reached

```
def pandas_train_input_fn(df):  # a Pandas dataframe
   return tf.estimator.inputs.pandas_input_fn(
       x = df,
       y = df['price'],
       batch_size=128,
       num_epochs=10,
       shuffle=True
   )

model.train(pandas_train_input_fn(df))
```

Trains until input exhausted (10 epochs) starting from checkpoint

# Training happens until input is exhausted or number of steps is reached

```
def pandas_train_input_fn(df):  # a Pandas dataframe
   return tf.estimator.inputs.pandas_input_fn(
       x = df,
       y = df['price'],
       batch_size=128,
       num_epochs=10,
       shuffle=True
   )

model.train(pandas_train_input_fn(df))

model.train(pandas_train_input_fn(df), steps=1000)
```

Trains until input exhausted (10 epochs) starting from checkpoint

1000 additional steps from checkpoint

# Training happens until input is exhausted or number of steps is reached

```
def pandas_train_input_fn(df):  # a Pandas dataframe
   return tf.estimator.inputs.pandas_input_fn(
       x = df,
       y = df['price'],
       batch_size=128,
       num_epochs=10,
       shuffle=True
   )

model.train(pandas_train_input_fn(df))

model.train(pandas_train_input_fn(df), steps=1000)

model.train(pandas_train_input_fn(df), max_steps=1000)
```

Trains until input exhausted (10 epochs) starting from checkpoint

1000 additional steps from checkpoint

1000 steps - might be nothing if checkpoint already there

# To add a new feature, add it to the list of feature columns and make sure it is present in data frame

```
featcols = [
   tf.feature_column.numeric_column("sq_footage"),
   tf.feature_column.categorical_column_with_vocabulary_list("type",
                                                             ["house", "apt"])

]

model = tf.estimator.LinearRegressor(featcols)

def train_input_fn(df):  # a Pandas dataframe
   return tf.estimator.inputs.pandas_input_fn(
       x = df,
       y = df['price'],
       batch_size=128, num_epochs=10, shuffle=True
   )

model.train(train_input_fn(df))
```

# To add a new feature, add it to the list of feature columns and make sure it is present in data frame

```
featcols = [
   tf.feature_column.numeric_column("sq_footage"),
   tf.feature_column.categorical_column_with_vocabulary_list("type",
                                                             ["house", "apt"])
   tf.feature_column.numeric_column("nbeds"),
]

model = tf.estimator.LinearRegressor(featcols)

def train_input_fn(df):  # a Pandas dataframe
   return tf.estimator.inputs.pandas_input_fn(
       x = df,
       y = df['price'],
       batch_size=128, num_epochs=10, shuffle=True
   )

model.train(train_input_fn(df))
```

# Lab

Implementing a Machine Learning model in TensorFlow using Estimator API

Google Cloud

# Train on large datasets with Dataset API

Lak Lakshmanan

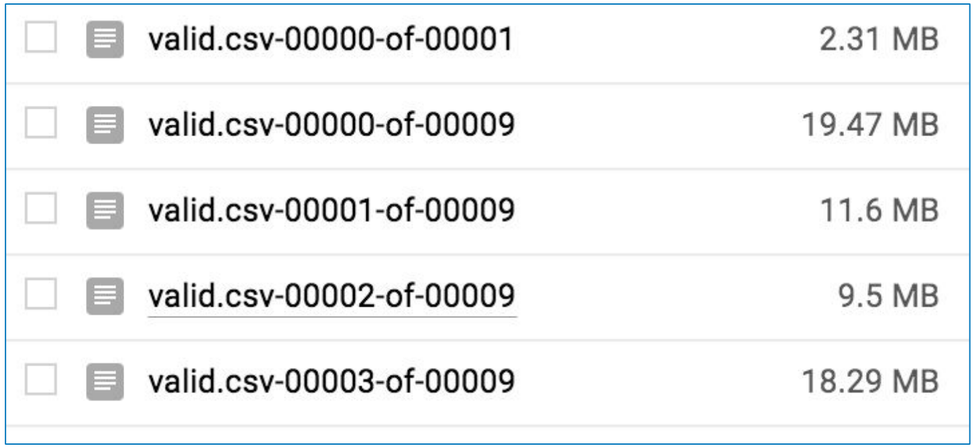
# Real World ML Models

| **Problem** | **Solution** |
| --- | --- |
| Out of memory data | ? |
| Distribution | ? |
| Need to evaluate during training | ? |
| Deployments that scale | ? |

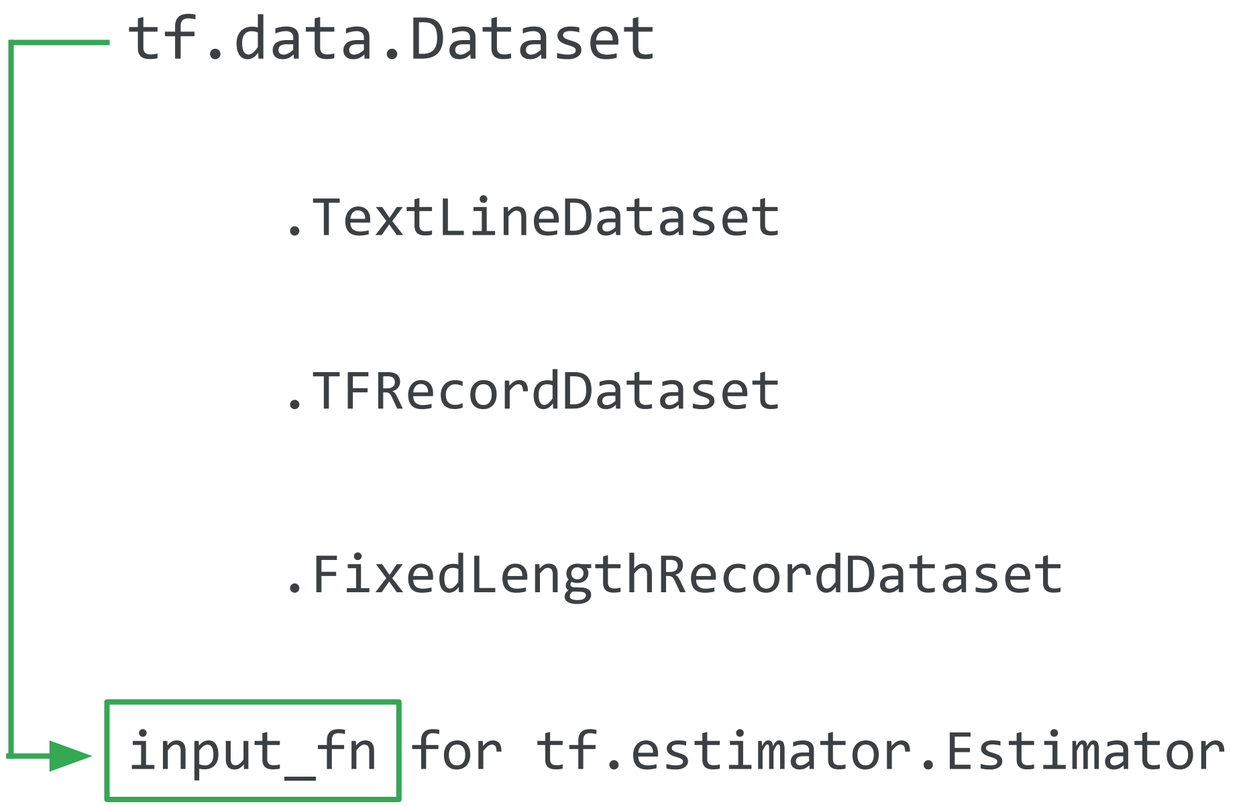
# Real World ML Models

| **Problem** | **Solution** |
| --- | --- |
| Out of memory data | Use the Dataset API |
| Distribution | ? |
| Need to evaluate during training | ? |
| Deployments that scale | ? |

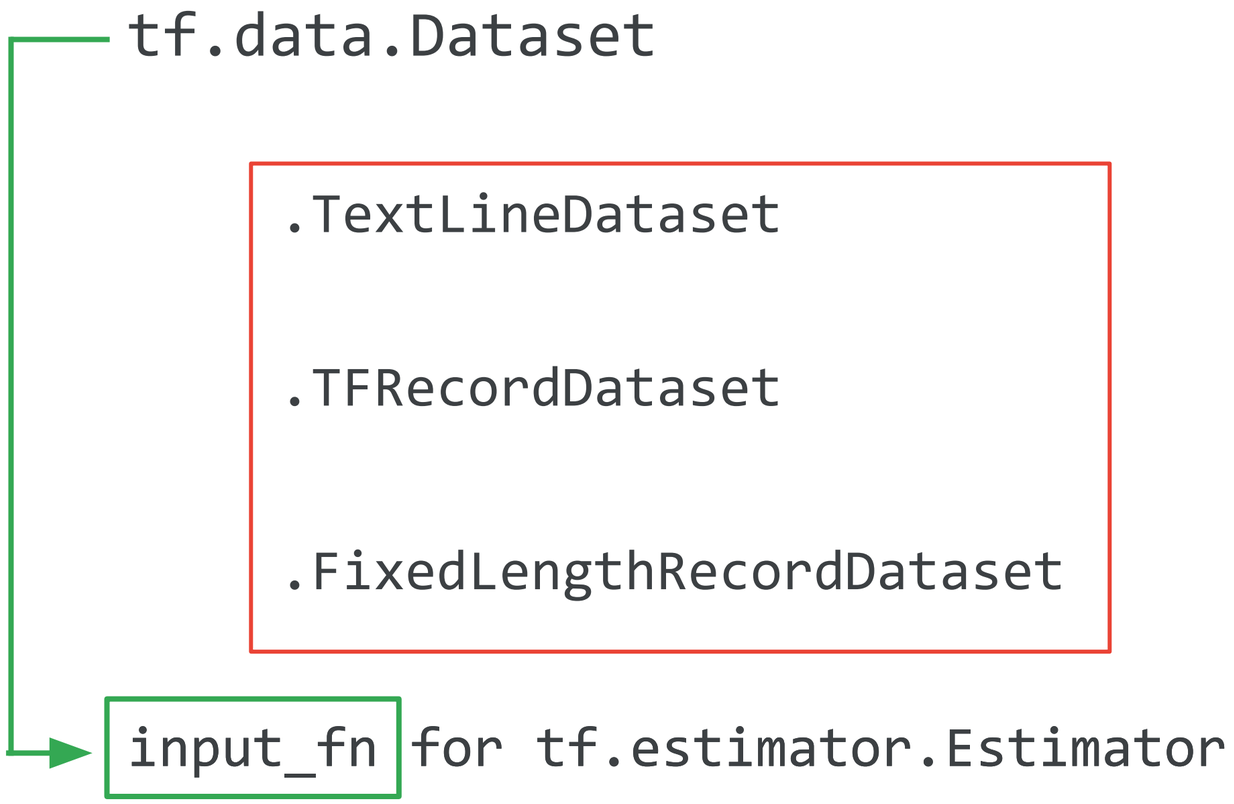
# Out-of-memory datasets tend to be sharded into multiple files

Datasets can be created from different file formats.
They generate input functions for Estimators.



Datasets can be created from different file formats.
They generate input functions for Estimators.



# Read one CSV file using TextLineDataset

```
def decode_line(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label

dataset = tf.data.TextLineDataset("train_1.csv") \
                 .map(decode_line)

dataset = dataset.shuffle(1000) \
                 .repeat(15)    \
                 .batch(128)

def input_fn():
   features, label = dataset.make_one_shot_iterator().get_next()
   return features, label

model.train(input_fn)
```

property type

sq_footage PRICE in K$

```
1001, house,  501
2001, house, 1001
3001, house, 1501
1001, apt,    701
2001, apt,   1301
3001, apt,   1901
1101, house,  526
2101, house, 1026
```

# Datasets handle shuffling, epochs, batching, ...

Shuffle buffer size

```
dataset = dataset.shuffle(1000) \
                 .repeat(15) \
                 .batch(128)
```

Nb of epochs

property type

| sq_footage | | PRICE in K$ |
|---|---|---|
| 1001, | house, | 501 |
| 2001, | house, | 1001 |
| 3001, | house, | 1501 |
| 1001, | apt, | 701 |
| 2001, | apt, | 1301 |
| 3001, | apt, | 1901 |
| 1101, | house, | 526 |
| 2101, | house, | 1026 |

# They support arbitrary transformations with `map()`

```
def decode_line(txt_line):
    cols = tf.decode_csv(txt_line, record_defaults=[[0],['house'],[0]])
    features = {'sq_footage': cols[0], 'type': cols[1]}
    label = cols[2] # price
    return features, label

dataset = tf.data.TextLineDataset("train_1.csv") \
                 .map(decode_line)
```

Dataset of text lines

property type

sq_footage PRICE in K$

```
1001, house,  501
2001, house, 1001
3001, house, 1501
1001, apt,    701
2001, apt,   1301
3001, apt,   1901
1101, house,  526
2101, house, 1026
```

# They support arbitrary transformations with `map()`

```
def decode_line(txt_line):
   cols = tf.decode_csv(txt_line, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label

dataset = tf.data.TextLineDataset("train_1.csv") \
                 .map(decode_line)
```

Dataset of pairs
(features, label)

property type

sq_footage PRICE in K$

```
1001, house,  501
2001, house, 1001
3001, house, 1501
1001, apt,    701
2001, apt,   1301
3001, apt,   1901
1101, house,  526
2101, house, 1026
```

# Datasets help create `input_fn`'s for Estimators

```
dataset = ...



def input_fn():
   features, label = dataset.make_one_shot_iterator().get_next()
   return features, label

model.train(input_fn)
```

property type

sq_footage PRICE in K$

```
1001, house,  501
2001, house, 1001
3001, house, 1501
1001, apt,    701
2001, apt,   1301
3001, apt,   1901
1101, house,  526
2101, house, 1026
```

# Datasets help create `input_fn`'s for Estimators

```
dataset = ...


def input_fn():
   features, label = dataset.make_one_shot_iterator().get_next()
   return features, label

model.train(input_fn)
```

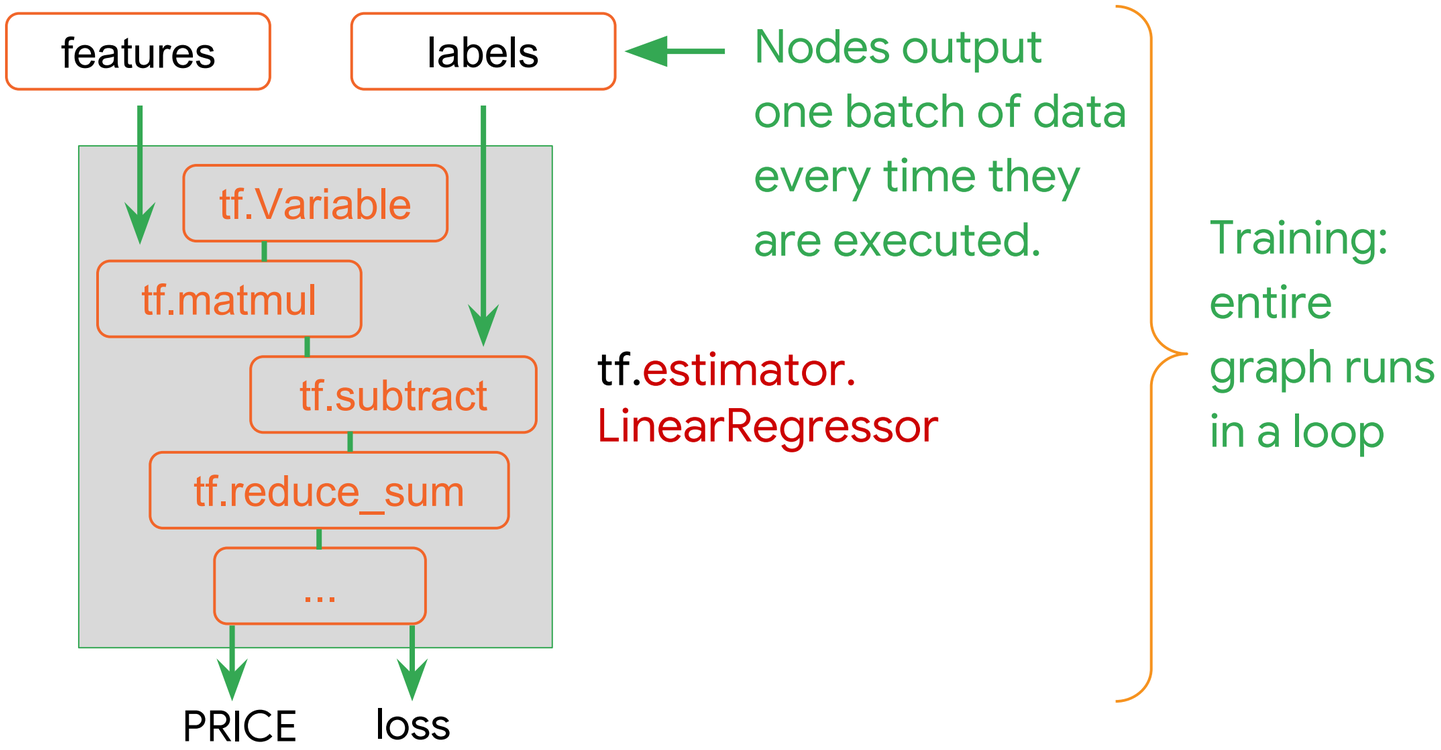
property type

sq_footage PRICE in K$

```
1001, house,  501
2001, house, 1001
3001, house, 1501
1001, apt,    701
2001, apt,   1301
3001, apt,   1901
1101, house,  526
2101, house, 1026
```

All the tf. commands that you write in Python **do not actually process any data**, they just build graphs

# Under the hood: what does an input function do ?

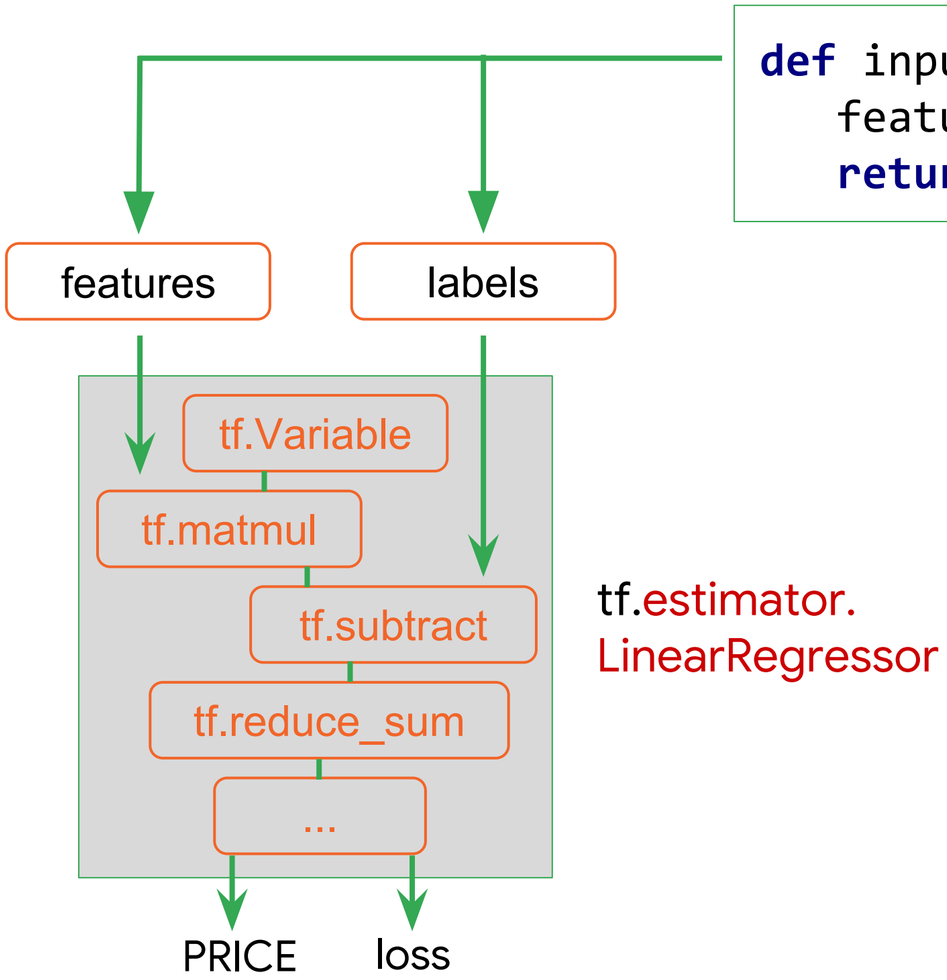



Nodes output one batch of data every time they are executed.

tf.estimator.
LinearRegressor

Training: entire graph runs in a loop

# Under the hood: what does an input function do ?



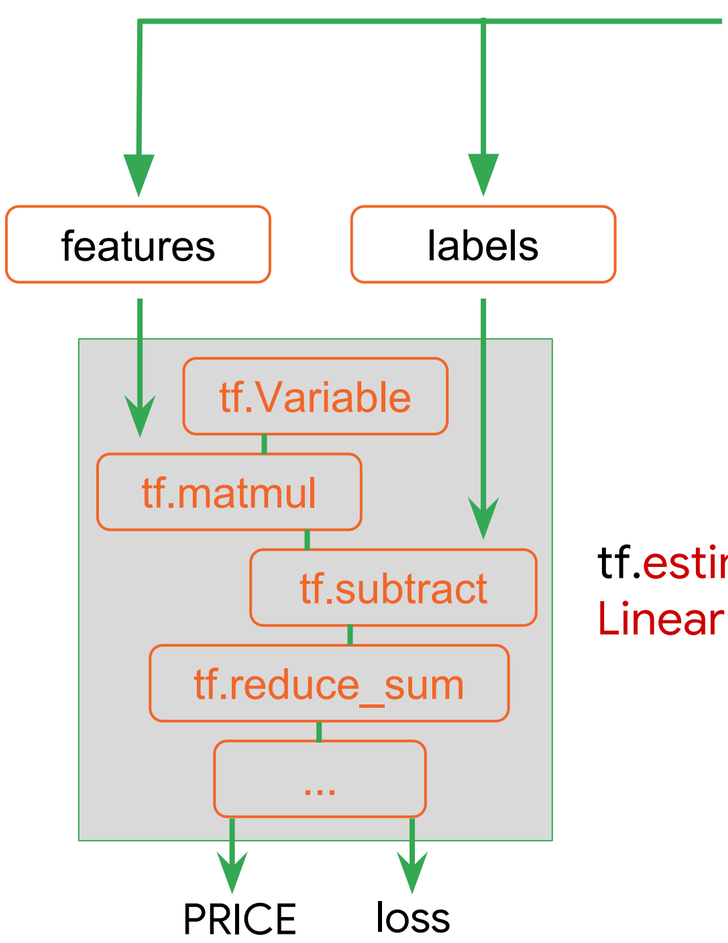

```
def input_fn():
   features, labels = ...
   return features, labels
```

Executed once

Returns Tensorflow nodes

tf.estimator.
LinearRegressor

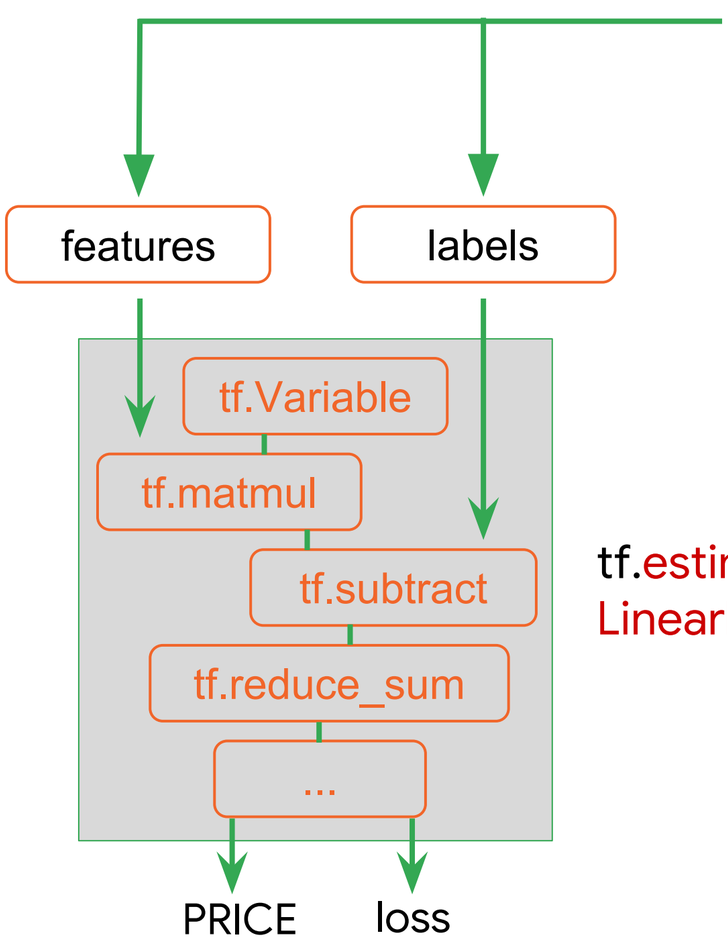
# Under the hood: Dataset API




```
def input_fn():
   features, label =
     dataset.make_one_shot_iterator().get_next()
   return features, labels
```

tf.estimator.
LinearRegressor

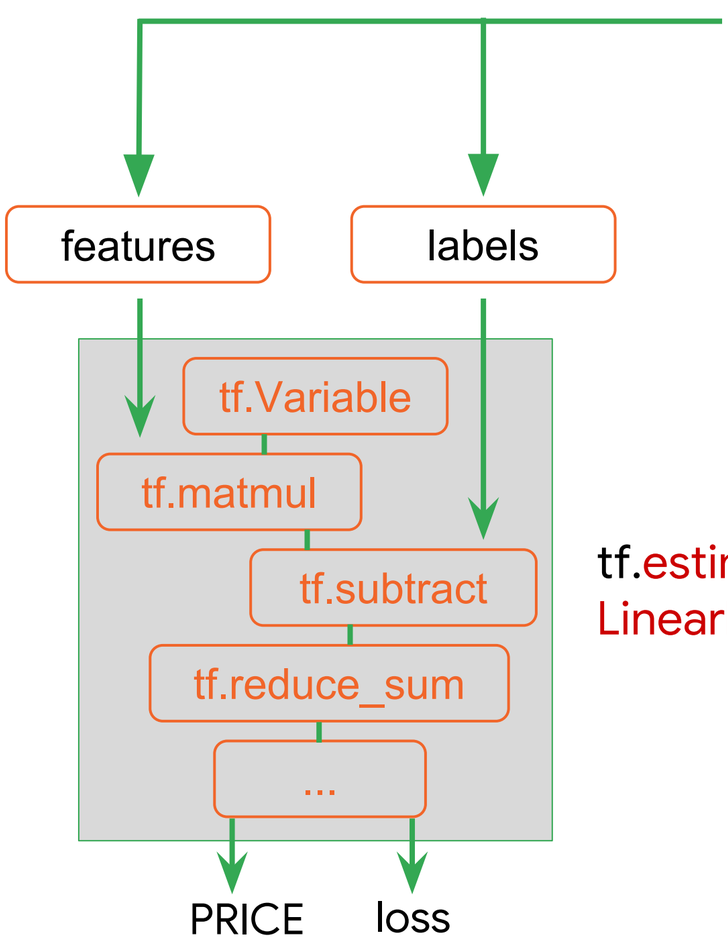
# Under the hood: Dataset API




```
def input_fn():
    features, label =
      dataset.make_one_shot_iterator().get_next()
    return features, labels
```

tf.estimator.
LinearRegressor

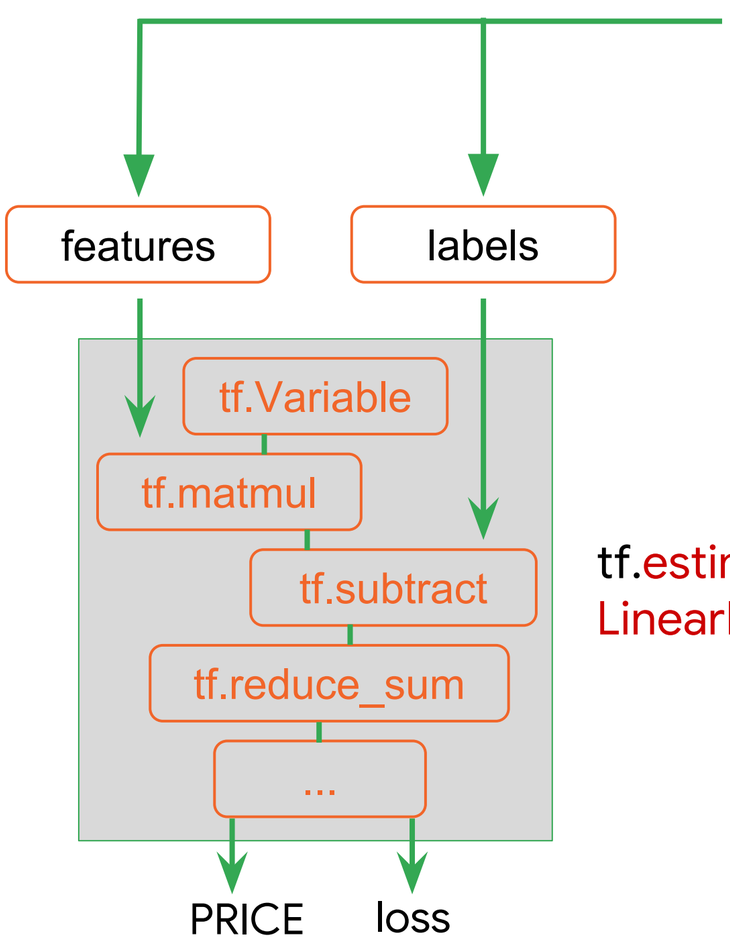
# Common Misconceptions about `input_fn`




```
def input_fn():
    features, label =
      dataset.make_one_shot_iterator().get_next()
    return features, labels
```

tf.estimator.
LinearRegressor

# Common Misconceptions about `input_fn`

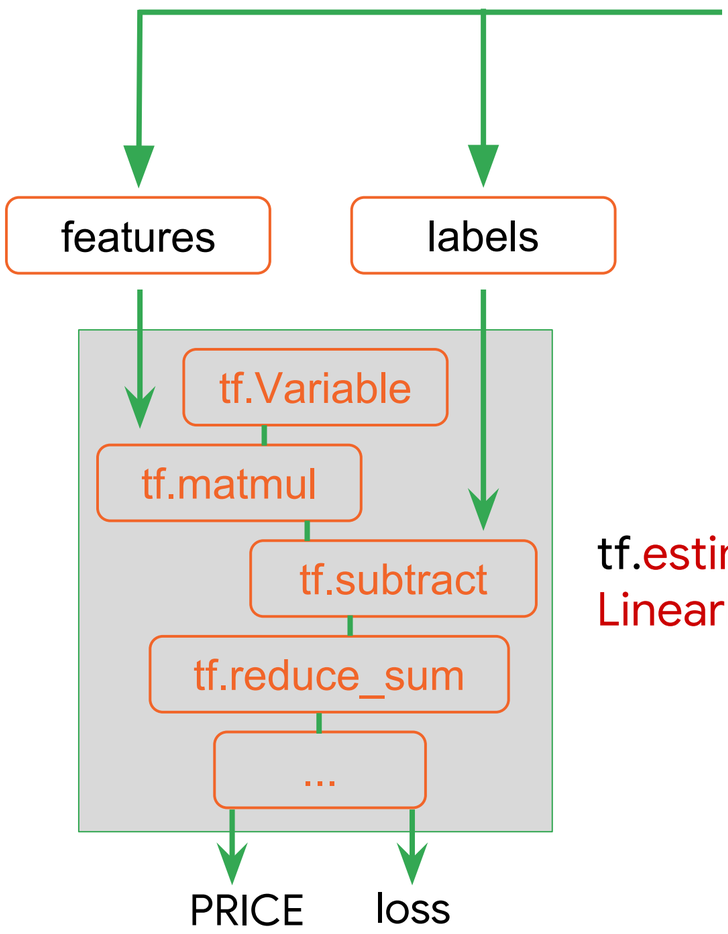



```
def input_fn():
   features, label =
     dataset.make_one_shot_iterator().get_next()
   return features, labels
```

1. Input functions called only once

tf.estimator.
LinearRegressor

# Common Misconceptions about `input_fn`




```
def input_fn():
   features, label =
     dataset.make_one_shot_iterator().get_next()
   return features, labels
```

tf.estimator.
LinearRegressor

1. Input functions called only once
2. Input functions return Tf nodes (not data)

# Read one CSV file using `TextLineDataset`

```
def decode_line(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label

dataset = tf.data.TextLineDataset("train_1.csv") \
                 .map(decode_line)

dataset = dataset.shuffle(1000) \
                 .repeat(15)    \
                 .batch(128)

def input_fn():
   features, label = dataset.make_one_shot_iterator().get_next()
   return features, label

model.train(input_fn)
```

property type

sq_footage PRICE in K$

```
1001, house,  501
2001, house, 1001
3001, house, 1501
1001, apt,    701
2001, apt,   1301
3001, apt,   1901
1101, house,  526
2101, house, 1026
```

# Read one CSV file using `TextLineDataset`

```
def decode_line(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label

dataset = tf.data.TextLineDataset("train_1.csv") \
                 .map(decode_line)

dataset = dataset.shuffle(1000) \
                 .repeat(15)    \
                 .batch(128)

def input_fn():
   features, label = dataset.make_one_shot_iterator().get_next()
   return features, label

model.train(input_fn)
```

property type

sq_footage PRICE in K$

```
1001, house,  501
2001, house, 1001
3001, house, 1501
1001, apt,    701
2001, apt,   1301
3001, apt,   1901
1101, house,  526
2101, house, 1026
```

# Read one CSV file using `TextLineDataset`

```
def decode_line(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label

dataset = tf.data.TextLineDataset("train_1.csv") \
                 .map(decode_line)

dataset = dataset.shuffle(1000) \
                 .repeat(15)    \
                 .batch(128)

def input_fn():
   features, label = dataset.make_one_shot_iterator().get_next()
   return features, label

model.train(input_fn)
```

property type

sq_footage PRICE in K$

```
1001, house,  501
2001, house, 1001
3001, house, 1501
1001, apt,    701
2001, apt,   1301
3001, apt,   1901
1101, house,  526
2101, house, 1026
```

# Read one CSV file using `TextLineDataset`

```
def decode_line(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label

dataset = tf.data.TextLineDataset("train_1.csv") \
                 .map(decode_line)

dataset = dataset.shuffle(1000) \
                 .repeat(15)    \
                 .batch(128)

def input_fn():
   features, label = dataset.make_one_shot_iterator().get_next()
   return features, label

model.train(input_fn)
```

property type

sq_footage PRICE in K$

```
1001, house,  501
2001, house, 1001
3001, house, 1501
1001, apt,    701
2001, apt,   1301
3001, apt,   1901
1101, house,  526
2101, house, 1026
```

# Read one CSV file using `TextLineDataset`

```
def decode_line(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label

dataset = tf.data.TextLineDataset("train_1.csv") \
                 .map(decode_line)

dataset = dataset.shuffle(1000) \
                 .repeat(15)    \
                 .batch(128)

def input_fn():
   features, label = dataset.make_one_shot_iterator().get_next()
   return features, label

model.train(input_fn)
```

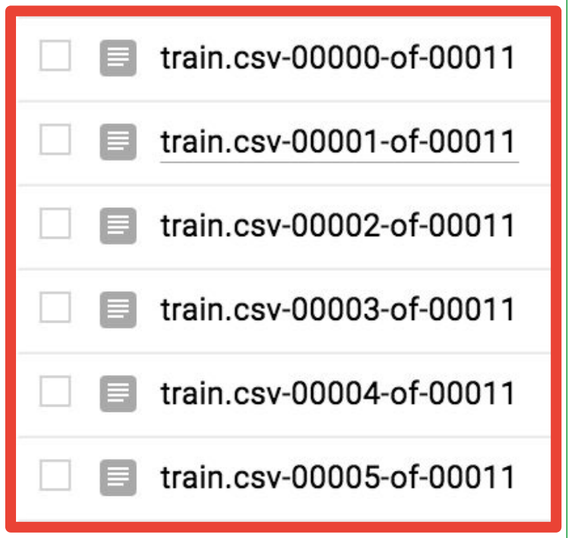
property type

sq_footage PRICE in K$

```
1001, house,  501
2001, house, 1001
3001, house, 1501
1001, apt,    701
2001, apt,   1301
3001, apt,   1901
1101, house,  526
2101, house, 1026
```

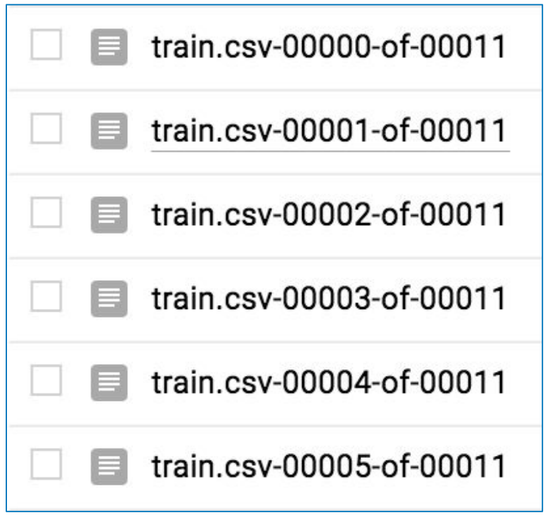
# Read a set of sharded CSV files using `TextLineDataset`

```
def decode_line(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label

dataset = tf.data.Dataset.list_files("train.csv-*")    \
                         .flat_map(tf.data.TextLineDataset) \
                         .map(decode_line)

dataset = dataset.shuffle(1000) \
                 .repeat(15)    \
                 .batch(128)

def input_fn():
   return dataset.make_one_shot_iterator().get_next()

model.train(input_fn)
```



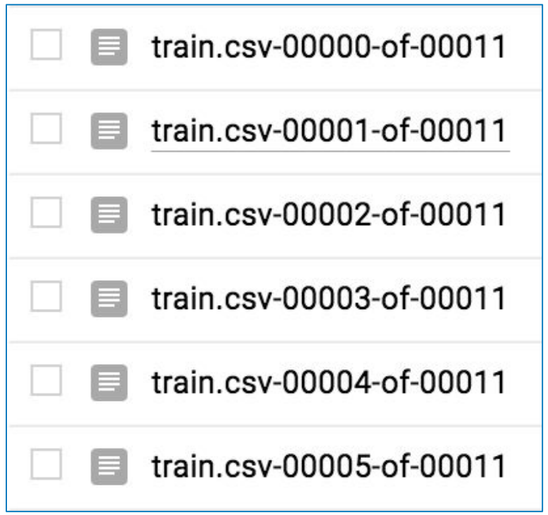
# Read a set of sharded CSV files using `TextLineDataset`

```
def decode_line(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label

dataset = tf.data.Dataset.list_files("train.csv-*")    \
                         .flat_map(tf.data.TextLineDataset) \
                         .map(decode_line)

dataset = dataset.shuffle(1000) \
                 .repeat(15)    \
                 .batch(128)

def input_fn():
   return dataset.make_one_shot_iterator().get_next()

model.train(input_fn)
```



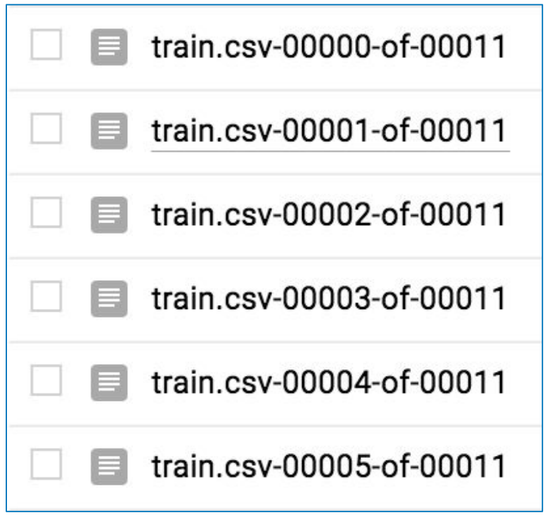
# Read a set of sharded CSV files using `TextLineDataset`

```
def decode_line(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label

dataset = tf.data.Dataset.list_files("train.csv-*")     \
                         .flat_map(tf.data.TextLineDataset) \
                         .map(decode_line)

dataset = dataset.shuffle(1000) \
                 .repeat(15)    \
                 .batch(128)

def input_fn():
   return dataset.make_one_shot_iterator().get_next()

model.train(input_fn)
```



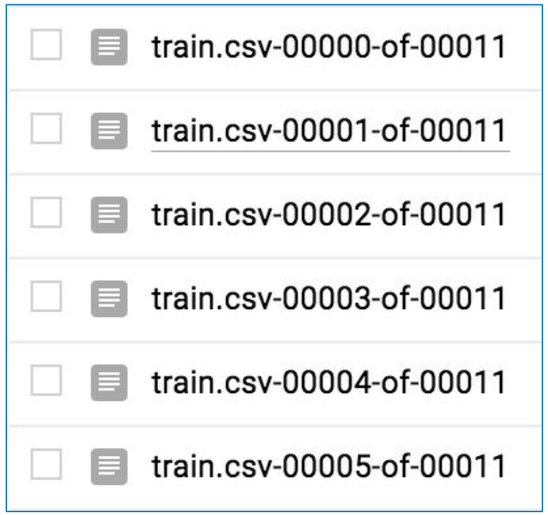
# Read a set of sharded CSV files using `TextLineDataset`

```
def decode_line(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label

dataset = tf.data.Dataset.list_files("train.csv-*")     \
                         .flat_map(tf.data.TextLineDataset) \
                         .map(decode_line)

dataset = dataset.shuffle(1000) \
                 .repeat(15)    \
                 .batch(128)

def input_fn():
   return dataset.make_one_shot_iterator().get_next()

model.train(input_fn)
```



# Read a set of sharded CSV files using `TextLineDataset`

```
def decode_line(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label

dataset = tf.data.Dataset.list_files("train.csv-*")    \
                         .flat_map(tf.data.TextLineDataset) \
                         .map(decode_line)

dataset = dataset.shuffle(1000) \
                 .repeat(15)    \
                 .batch(128)

def input_fn():
   return dataset.make_one_shot_iterator().get_next()

model.train(input_fn)
```



# The real benefit of Dataset is that you can do more than just ingest data

```
dataset =
tf.data.TextLineDataset(filename)\
          .skip(num_header_lines)\
          .map(add_key)\
          .map(decode_csv)\
          .map(lambda feats,
               labels: preproc(feats), labels)
          .filter(is_valid)\
          .cache()
```

# LAB

Scaling up TensorFlow ingest using batching

Google Cloud

# Big Jobs, Distributed training

Martin Gorner

# Real World ML Models

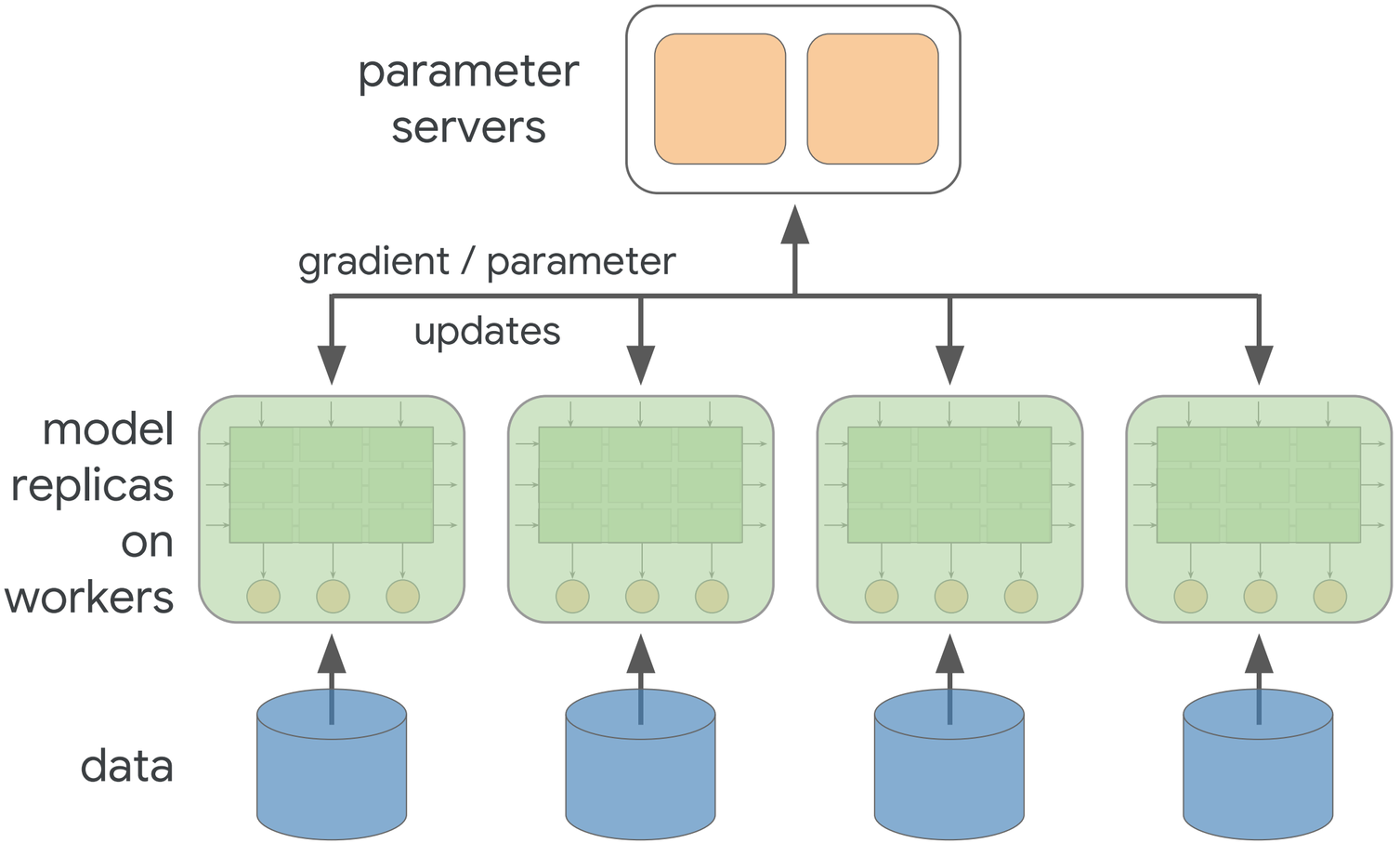
| Problem | Solution |
| --- | --- |
| Out of memory data | Use the Dataset API |
| Distribution | Use `train_and_evaluate` |
| Need to evaluate during training | ? |
| Deployments that scale | ? |

# `estimator.train_and_evaluate` is the preferred method for training real-world models

```
estimator = tf.estimator.LinearRegressor(...)

tf.estimator.train_and_evaluate(estimator, ...)
```

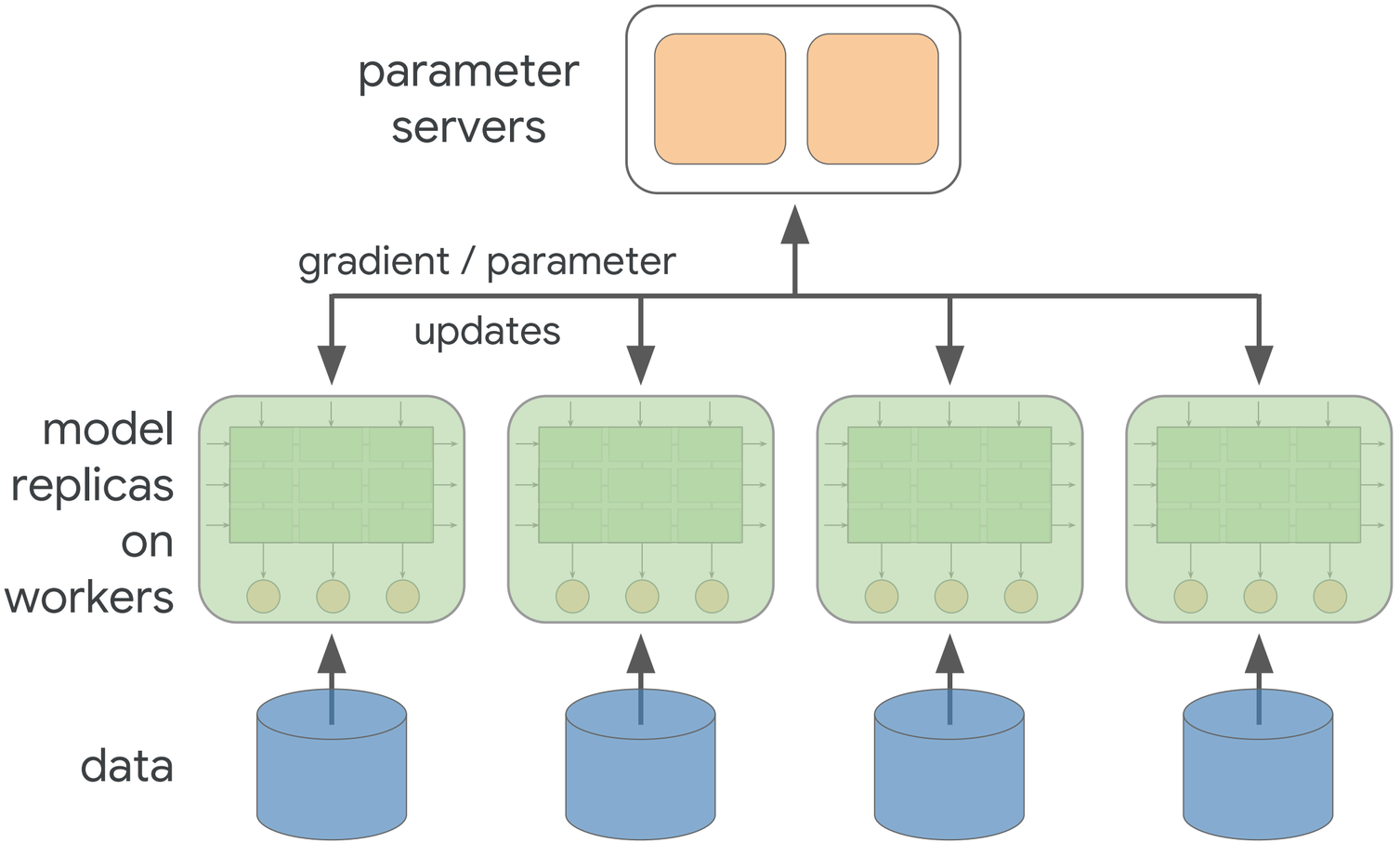
**data parallelism** =
replicate your model on
multiple workers

# Distributed training using “data parallelism”




- Distribute the graph
- Share variables
- Manage workers
- Handle exceptions / failures
- Create checkpoint files

# Distributed training using “data parallelism”




- Distribute the graph
- Share variables
- Manage workers
- Handle exceptions / failures
- Create checkpoint files

For free!

→ `tf.estimator.train_and_evaluate`

# `estimator.train_and_evaluate` is the preferred method for training real-world models

```
estimator = tf.estimator.LinearRegressor(
                      feature_columns=featcols,
                      config=run_config)

...

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

You need:

1. Estimator
2. Run Config
3. Train Spec
4. Eval Spec

# run_config tells the estimator where and how often to write Checkpoints and Tensorboard logs ("summaries")

```
run_config = tf.estimator.RunConfig(
                           model_dir=output_dir,
                           save_summary_steps=100,
                           save_checkpoints_steps=2000)

estimator = tf.estimator.LinearRegressor(config=run_config, ...)
```

# The TrainSpec tells the estimator how to get training data

Use Datasets

```
train_spec =
tf.estimator.TrainSpec(input_fn=train_input_fn, max_steps=50000)

...

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

# The EvalSpec controls the evaluation and the checkpointing of the model since they happen at the same time

```
eval_spec =
tf.estimator.EvalSpec(
           input_fn=eval_input_fn,
           steps=100, # evals on 100 batches
           throttle_secs=600 # eval no more than every 10 min
           exporters=...)

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

# The EvalSpec controls the evaluation and the checkpointing of the model since they happen at the same time

Use Datasets

```
eval_spec =
tf.estimator.EvalSpec(
           input_fn=eval_input_fn,
           steps=100, # evals on 100 batches
           throttle_secs=600 # eval no more than every 10 min
           exporters=...)

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

# The EvalSpec controls the evaluation and the checkpointing of the model since they happen at the same time

```
eval_spec =
tf.estimator.EvalSpec(
           input_fn=eval_input_fn,
           steps=100, # evals on 100 batches
           throttle_secs=600 # eval no more than every 10 min
           exporters=...)

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

# Recap with all the code

```
run_config =
tf.estimator.RunConfig(model_dir=output_dir, ...)

estimator =
tf.estimator.LinearRegressor(featcols, config=run_config)

train_spec =
tf.estimator.TrainSpec(input_fn=train_input_fn, max_steps=1000)

export_latest =
tf.estimator.LatestExporter(serving_input_receiver_fn=serving_input_fn)

eval_spec =
tf.estimator.EvalSpec(input_fn=eval_input_fn, exporters=export_latest)


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

# Recap with all the code

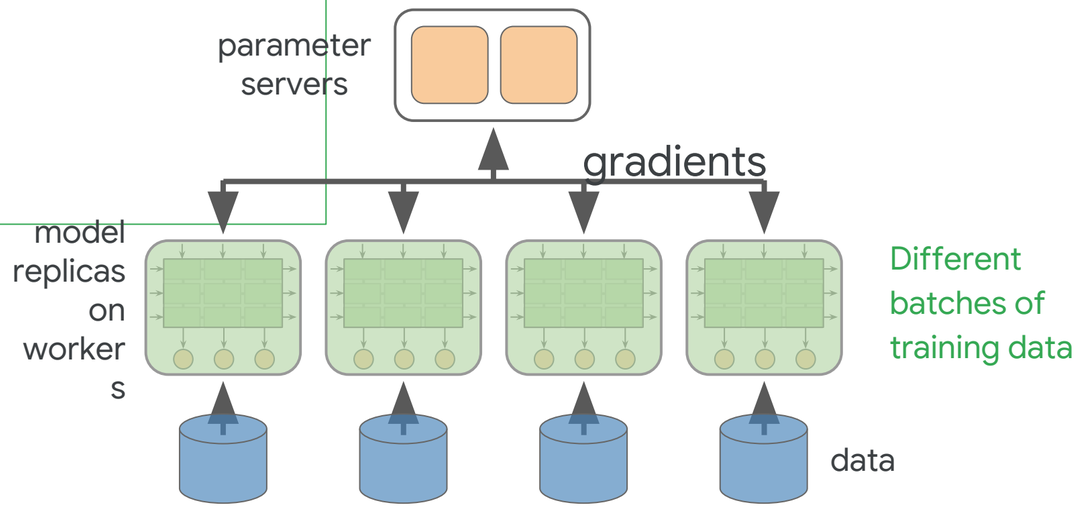
```
run_config =
tf.estimator.RunConfig(model_dir=output_dir, ...)

estimator =
tf.estimator.LinearRegressor(featcols, config=run_config)

train_spec =
tf.estimator.TrainSpec(input_fn=train_input_fn, max_steps=1000)

export_latest =
tf.estimator.LatestExporter(serving_input_receiver_fn=serving_input_fn)

eval_spec =
tf.estimator.EvalSpec(input_fn=eval_input_fn, exporters=export_latest)

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

# Shuffling is even more important in distributed training

```
dataset =
tf.data.Dataset.list_files("train.csv-*")  \
        .shuffle(100)                      \
        .flat_map(tf.data.TextLineDataset) \
        .map(decode_csv)

dataset = dataset.shuffle(1000) \
                 .repeat(15)    \
                 .batch(128)
```

# Shuffling is even more important in distributed training.

```
dataset =
tf.data.Dataset.list_files("train.csv-*")  \
        .shuffle(100)                      \
        .flat_map(tf.data.TextLineDataset) \
        .map(decode_csv)

dataset = dataset.shuffle(1000) \
                 .repeat(15)    \
                 .batch(128)
```



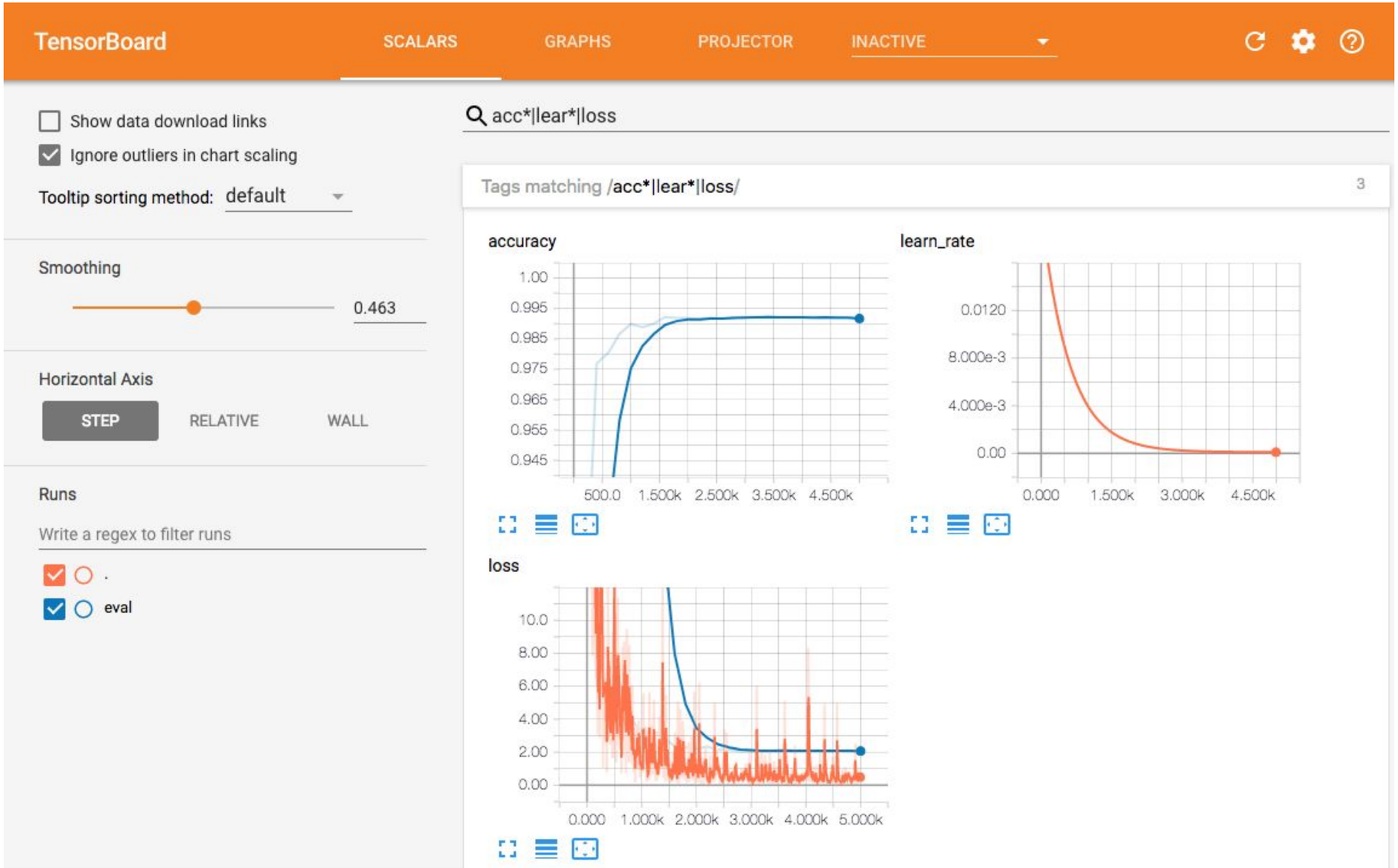
Google Cloud

# Monitoring with TensorBoard

Martin Gorner

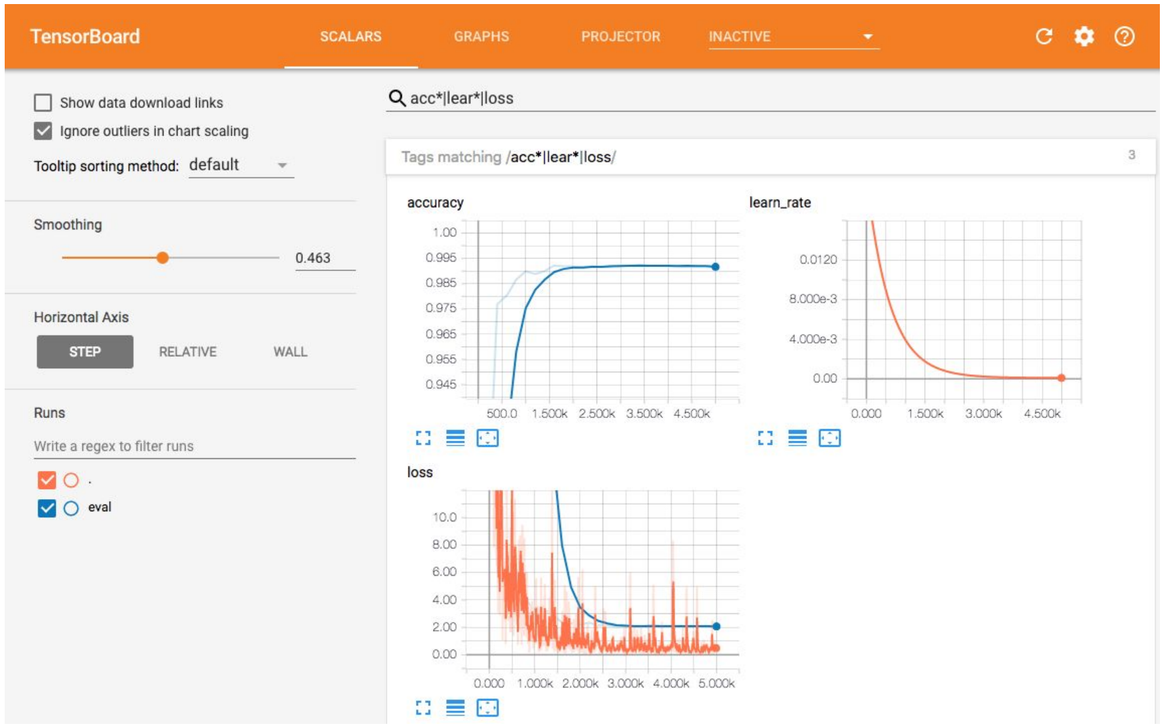
# Real World ML Models

| Problem | Solution |
| --- | --- |
| ✓ Out of memory data | Use the Dataset API |
| ✓ Distribution | Use `train_and_evaluate` |
| Need to evaluate during training | Use `train_and_evaluate` + TensorBoard |
| Deployments that scale | ? |

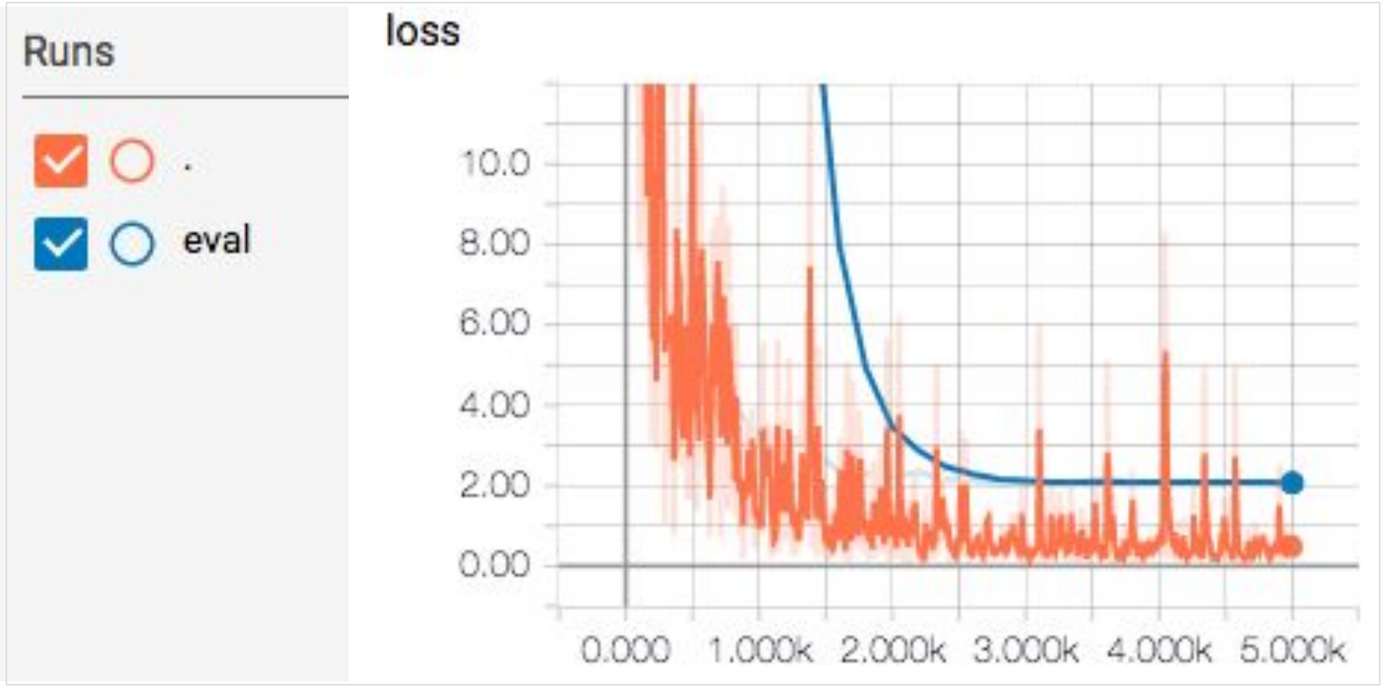

TensorBoard
SCALARS
GRAPHS
PROJECTOR
INACTIVE
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0.463
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
.
eval
acc*|lear*|loss
Tags matching /acc*|lear*|loss/
3
accuracy
1.00
0.995
0.985
0.975
0.965
0.955
0.945
500.0
1.500k
2.500k
3.500k
4.500k
learn_rate
0.0120
8.000e-3
4.000e-3
0.00
0.000
1.500k
3.000k
4.500k
loss
10.0
8.00
6.00
4.00
2.00
0.00
0.000
1.000k
2.000k
3.000k
4.000k
5.000k

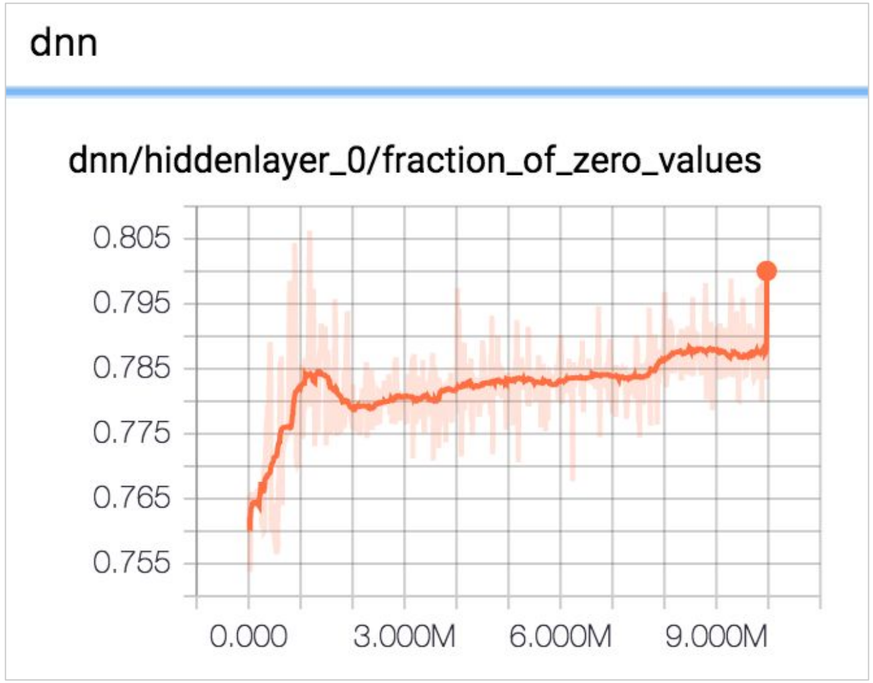
# Point Tensorboard to your output directory and the dashboards appear in your browser at localhost:6006

```
tf.estimator.RunConfig(model_dir='output_dir’)
> tensorboard --logdir output_dir
```

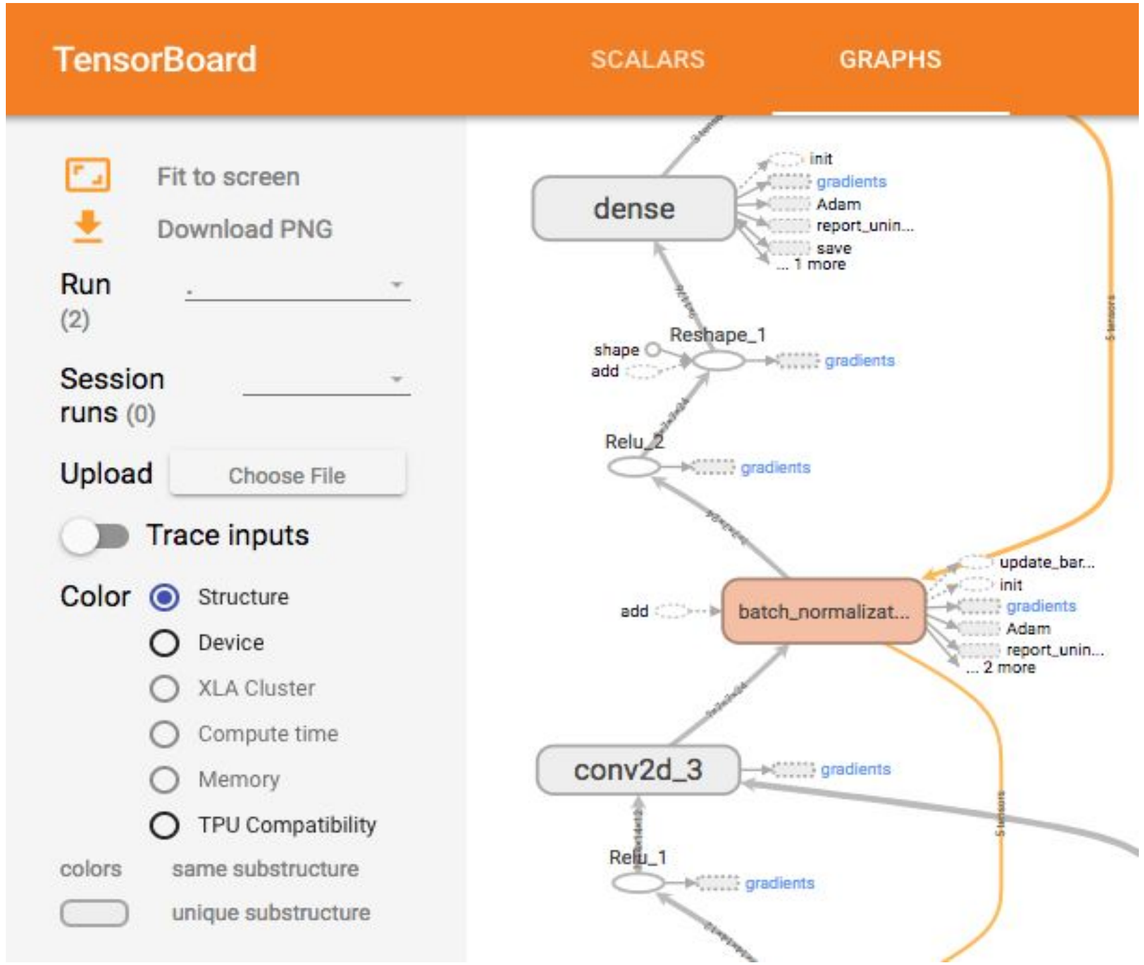


Pre-made Estimators export relevant metrics, embeddings, histograms, etc. for TensorBoard, so there is nothing more to do



# The dashboard for the graph



# If you are writing a custom Estimator model, you can add summaries for Tensorboard with a single line

```
tf.summary.scalar('meanVar1', tf.reduce_mean(var1))

...

tf.summary.text('outClass', stringvar))
```

these names will show up in TensorBoard

Sprinkle appropriate summary ops throughout your code:

`tf.summary.scalar`

`tf.summary.image`

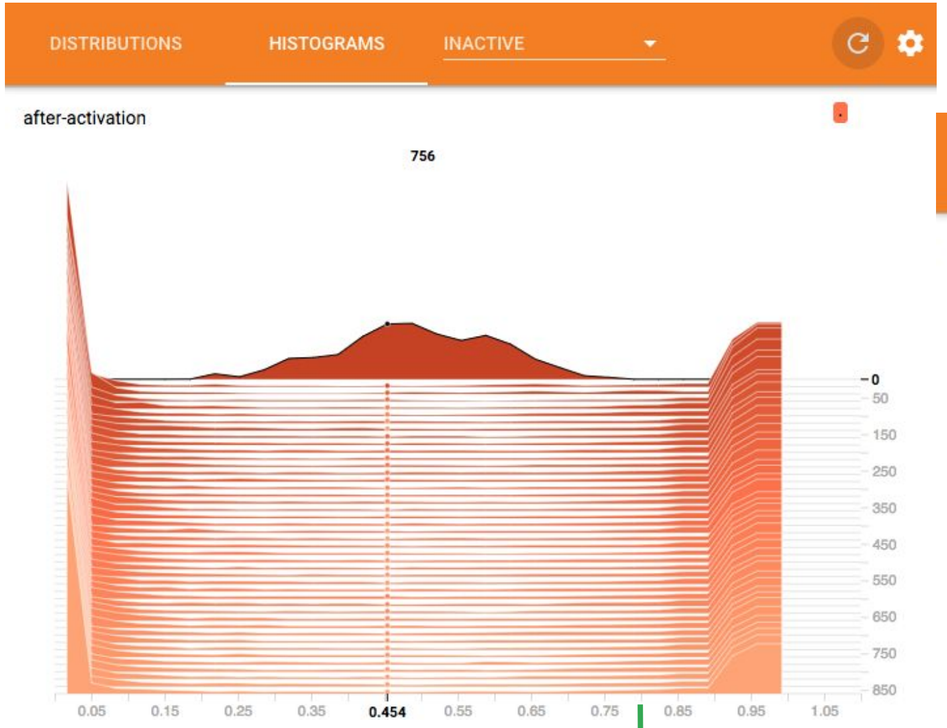
`tf.summary.audio`

`tf.summary.text`

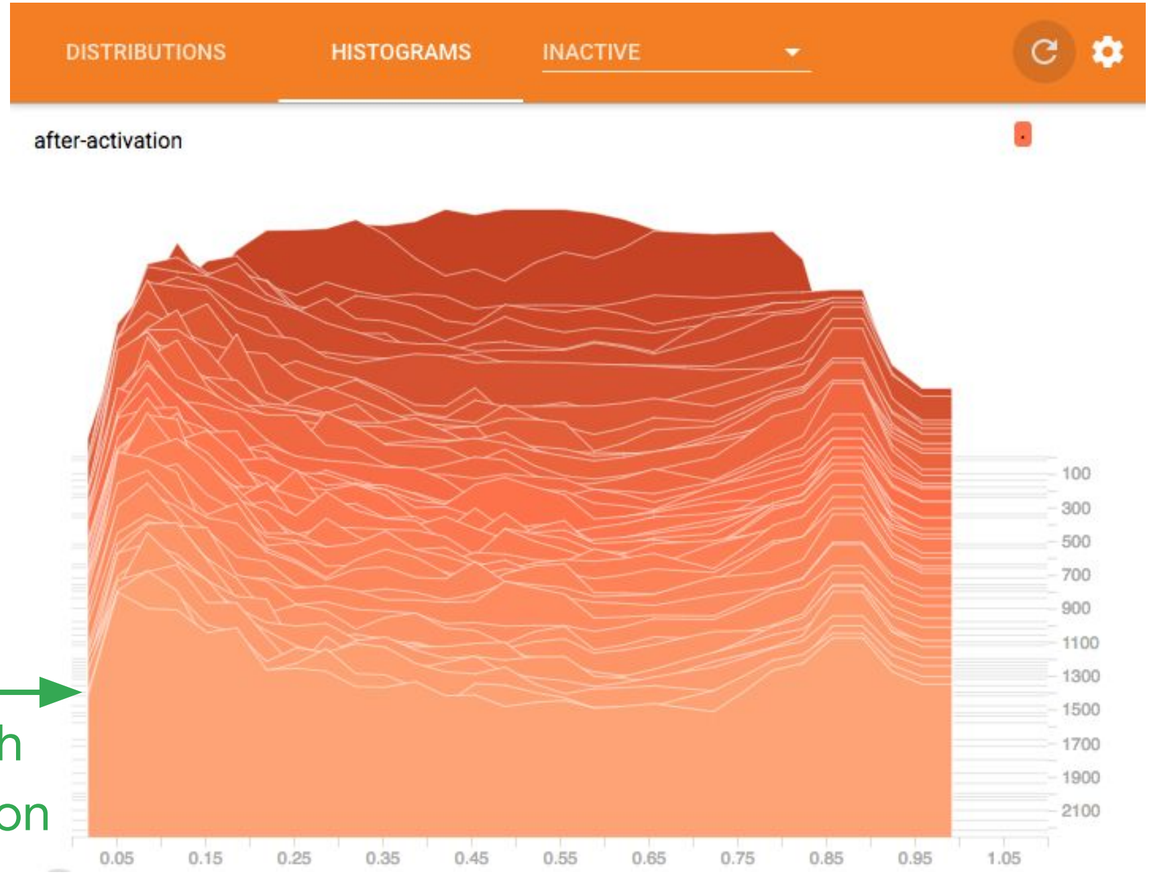
`tf.summary.histogram`

https://www.tensorflow.org/get_started/summaries_and_tensorboard

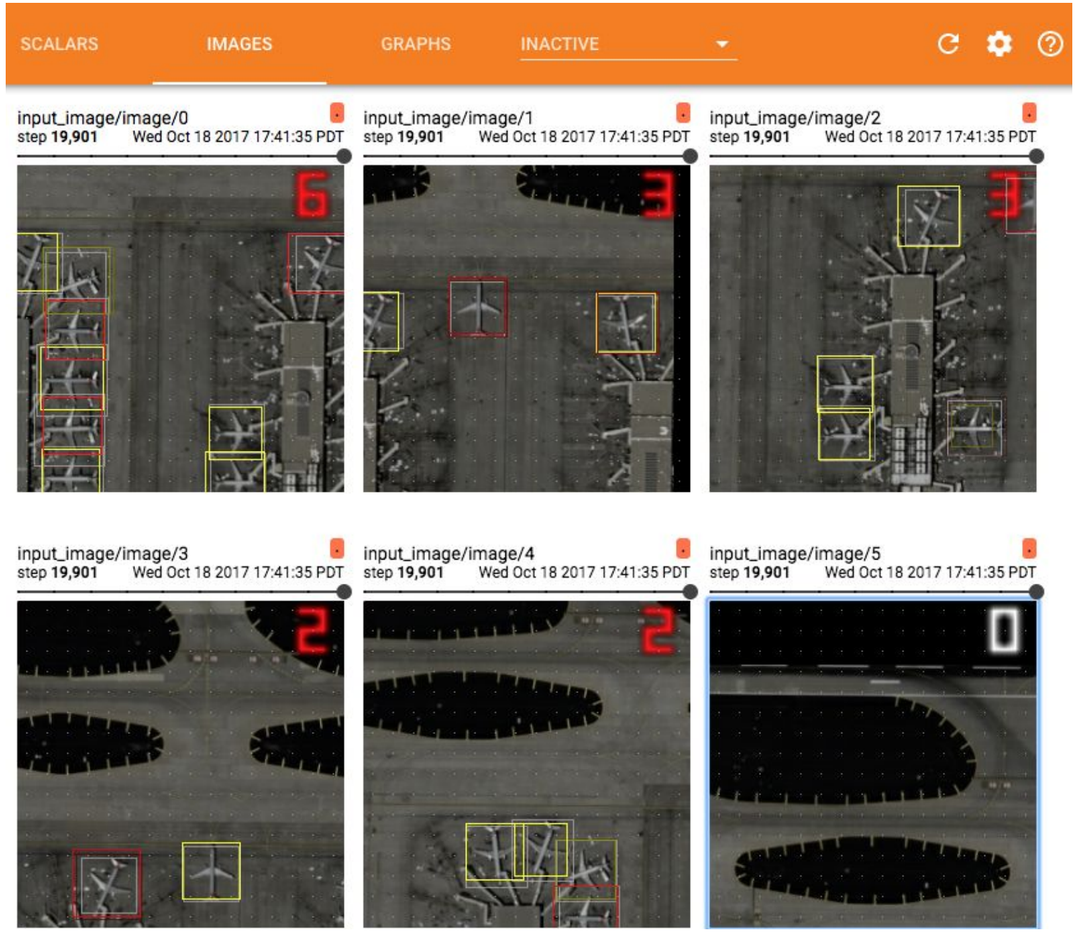
# The dashboard for histograms




After Batch Normalization



# Images and Audio have their own (non-scalar) summary ops

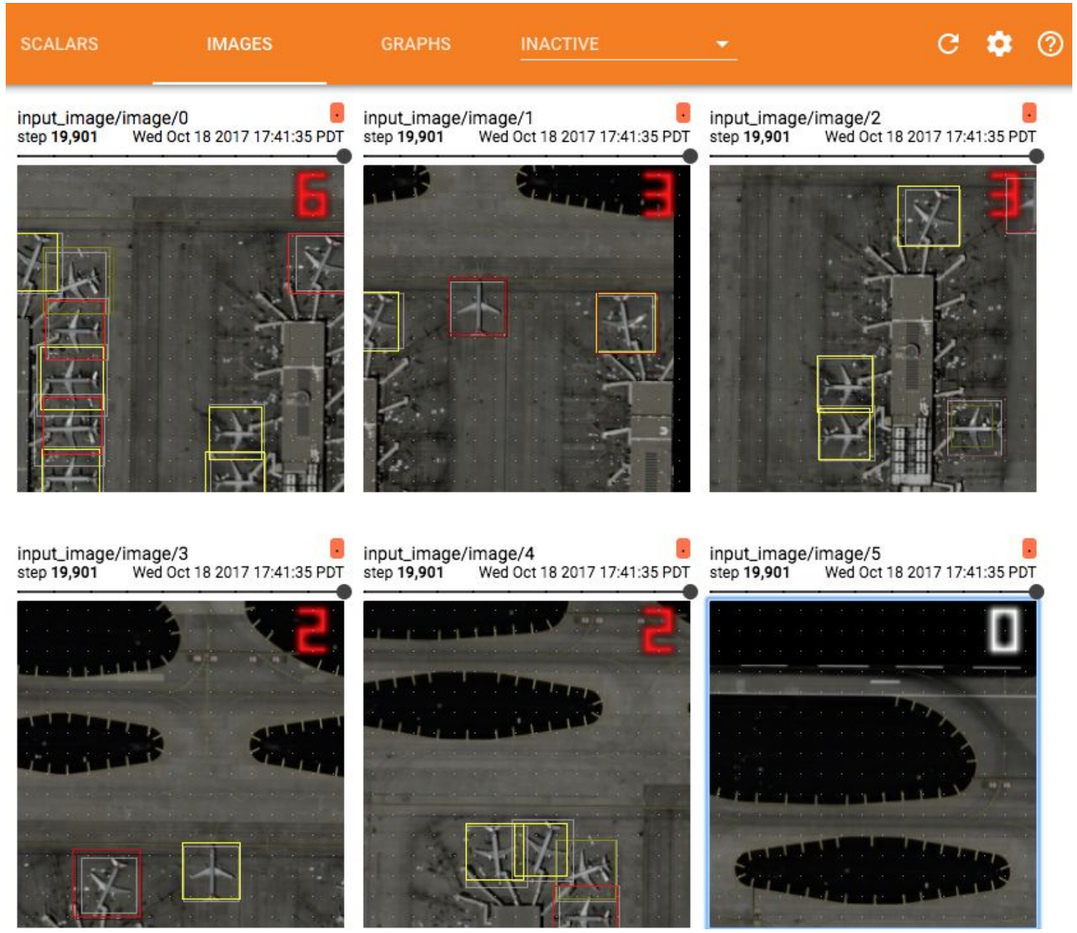


# Images and Audio have their own (non-scalar) summary ops

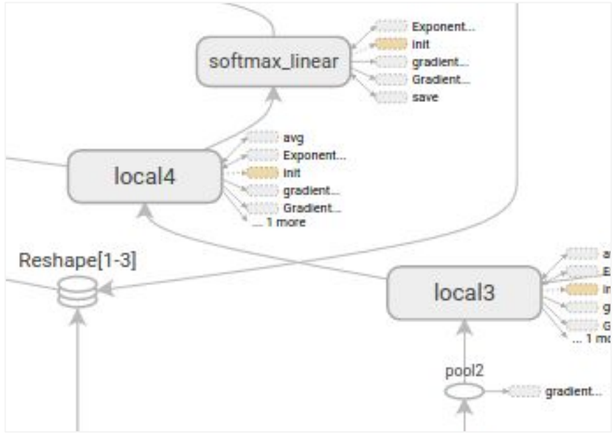
```
p = tf.placeholder("uint8", (None, ht, wd,num_channels))
s = tf.summary.image(“im1”, p)
```

```
p = tf.placeholder("float32",
            (None, duration_frames, num_channels))
s = tf.summary.audio(name='au1', tensor=p, sample_rate=4000)
```

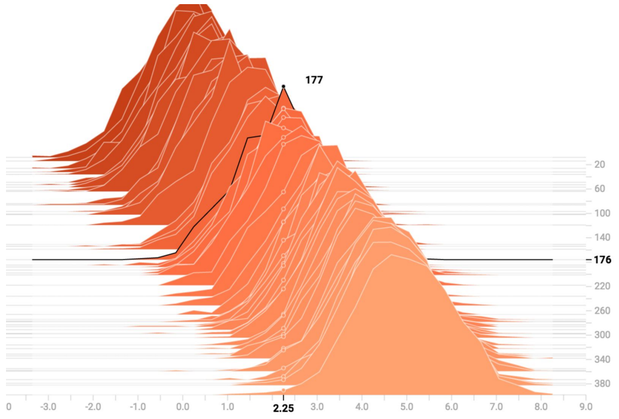
# Images and Audio have their own (non-scalar) summary ops



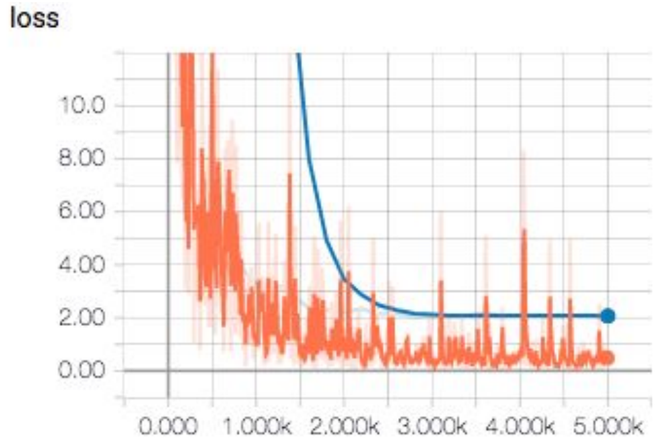
# TensorBoard has a suite of visualization tools to explore and explain your model and results




Graph Explorer




Histogram Dashboard




Scalar Dashboard

Many more including Audio, Image and Text Dashboards ...

# Real World ML Models

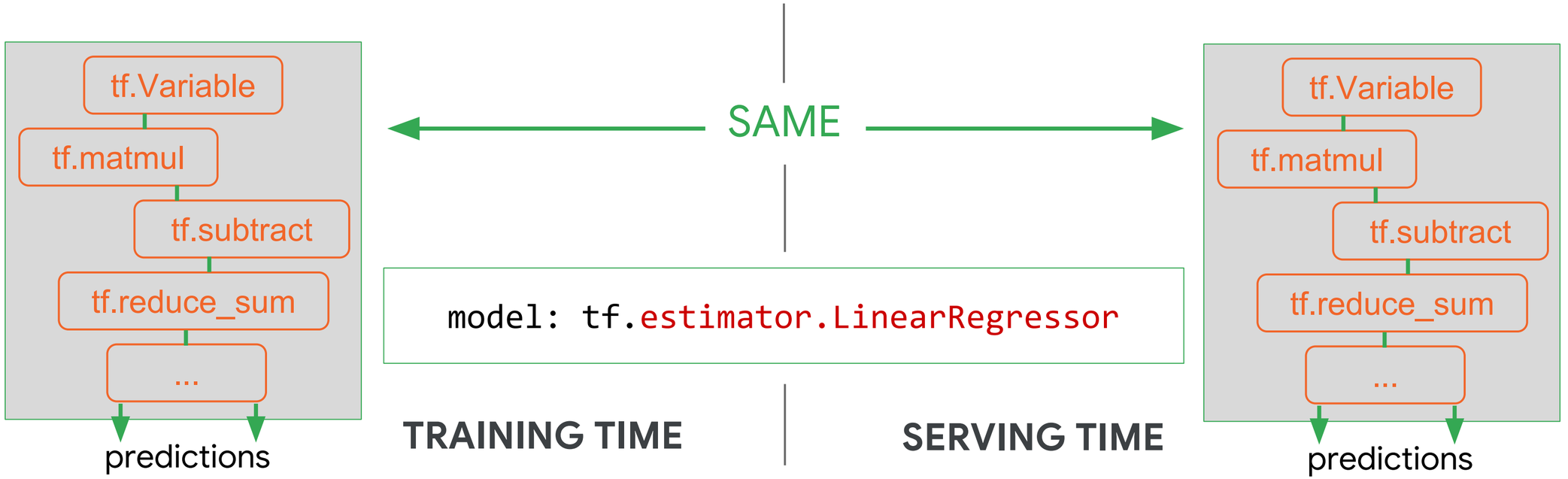
| Problem | Solution |
| --- | --- |
| Out of memory data | Use the Dataset API |
| Distribution | Use `train_and_evaluate` |
| Need to evaluate during training | Use `train_and_evaluate` + TensorBoard |
| Deployments that scale | Use serving input function |

# Recap with all the code

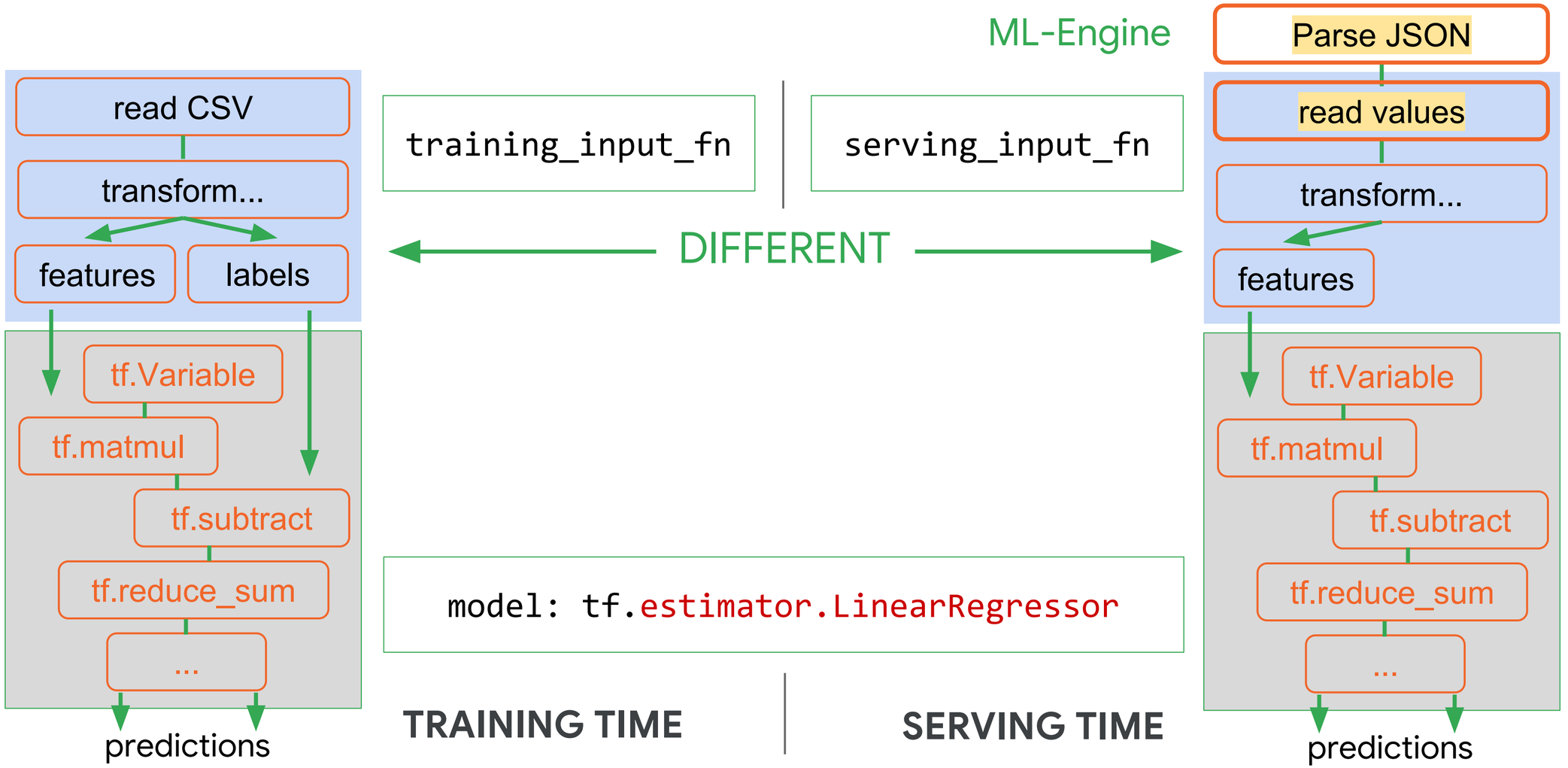
```
run_config =
tf.estimator.RunConfig(model_dir=output_dir, ...)

estimator =
tf.estimator.LinearRegressor(featcols, config=run_config)

train_spec =
tf.estimator.TrainSpec(input_fn=train_input_fn, max_steps=1000)

export_latest =
tf.estimator.LatestExporter(serving_input_receiver_fn=serving_input_fn)

eval_spec =
tf.estimator.EvalSpec(input_fn=eval_input_fn, exporters=export_latest)


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

Serving and training-time inputs are often very different

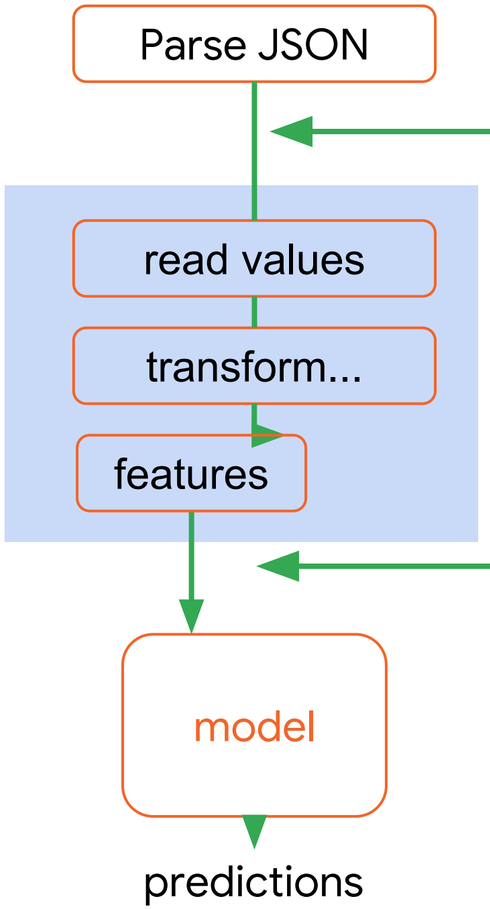
# Serving and training-time inputs are often very different



# Serving and training-time inputs are often very different



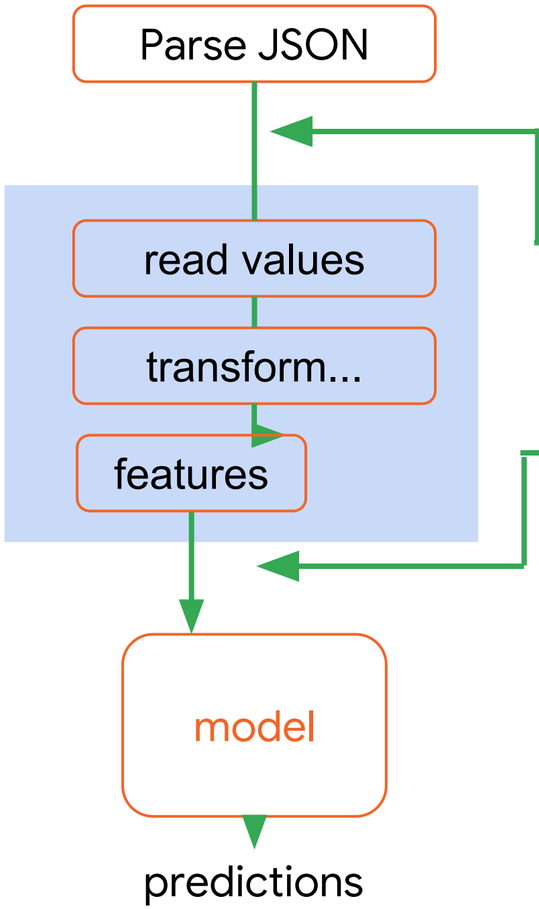
# Serving input function transforms from parsed JSON data to the data your model expects




Batch size

```
def serving_input_fn():
    json = { 'sq_footage' : tf.placeholder(tf.int32, [None]),
             'prop_type'  : tf.placeholder(tf.string, [None])
           }
    # ... transformations ...
    features = { 'sq_footage': json['sq_footage'],
                 'type': json['prop_type'],
               }
    return tf.estimator.export.ServingInputReceiver(features,
                                                    json)


exp = tf.estimator.LatestExporter("pricing",
serving_input_fn)
eval_spec = tf.estimator.EvalSpec(exporters=exp, ...)
```

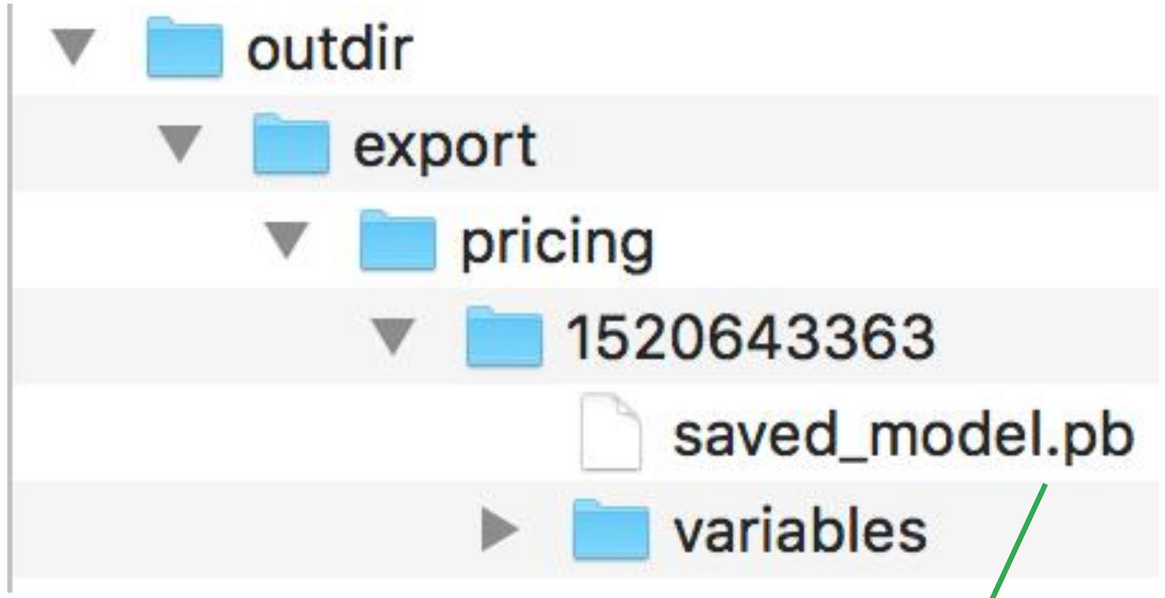
# Serving input function transforms from parsed JSON data to the data your model expects




Batch size

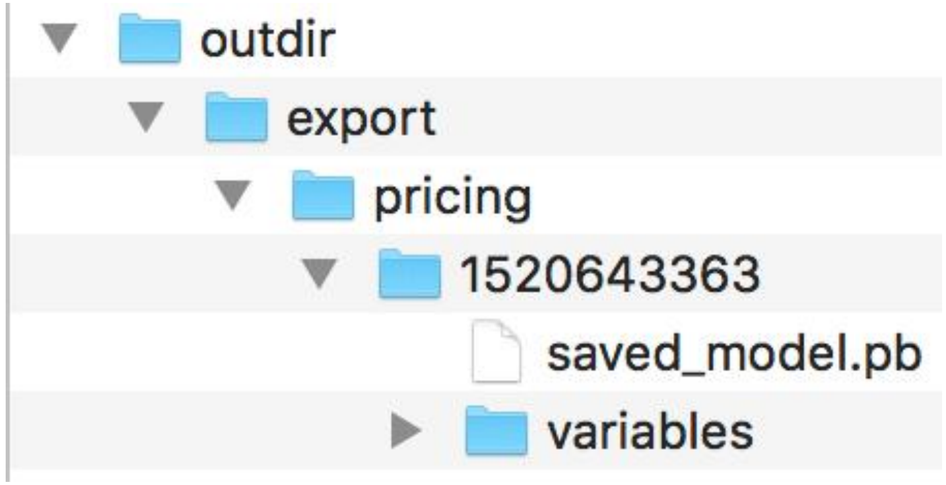
```
def serving_input_fn():
    json = { 'sq_footage' : tf.placeholder(tf.int32, [None]),
             'prop_type'  : tf.placeholder(tf.string, [None])
           }
    # ... transformations ...
    features = { 'sq_footage': json['sq_footage'],
                 'type': json['prop_type'],
               }
    return tf.estimator.export.ServingInputReceiver(features,
                                                    json)

exp = tf.estimator.LatestExporter("pricing",
serving_input_fn)
eval_spec = tf.estimator.EvalSpec(exporters=exp, ...)
```

# The exported model is ready to deploy




One click deploy to ML Engine

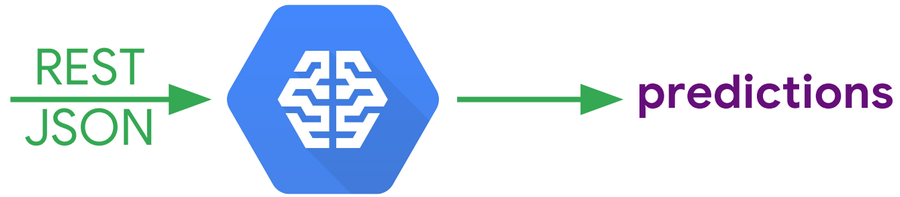
# The exported model is ready to deploy



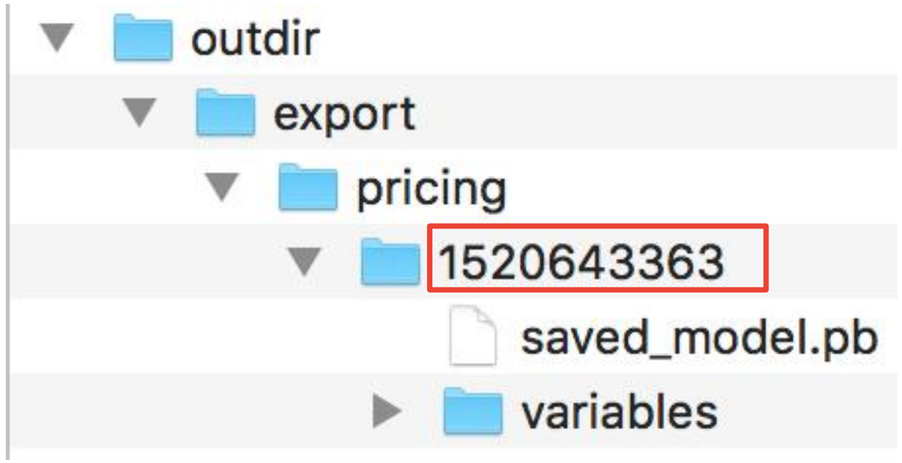

```
gcloud ml-engine predict
--model <model_name>
--json-instances data.json
```

Online predictions :

```
{"instances":[
 {"sq_footage": 1520, "prop_type": "apt"},
 {"sq_footage": 1520, "prop_type": "house"}
]}
```



# The exported model is ready to deploy




To test locally :

```
{"sq_footage": 1520, "prop_type": "apt"},
{"sq_footage": 1520, "prop_type": "house"}
```

model

```
gcloud ml-engine local predict \
   --model-dir outdir/export/pricing/1520643363 \
   --json-instances test_jsons.txt
```

# Example serving input function that decodes JPEGs

```
def serving_input_fn():
    json = {'jpeg_bytes': tf.placeholder(tf.string, [None])}

    def decode(jpeg):
        pixels = tf.image.decode_jpeg(jpeg, channels=3)
        return pixels

    pics = tf.map_fn(decode, json['jpeg_bytes'], dtype=tf.uint8)

    features = {'pics': pics}
    return tf.estimator.export.ServingInputReceiver(features, json)
```

Output shape:
[batch, width, height, 3]

# Example serving input function that decodes JPEGs

```
def serving_input_fn():
    json = {'jpeg_bytes': tf.placeholder(tf.string, [None])}

    def decode(jpeg):
        pixels = tf.image.decode_jpeg(jpeg, channels=3)
        return pixels

    pics = tf.map_fn(decode, json['jpeg_bytes'], dtype=tf.uint8)

    features = {'pics': pics}
    return tf.estimator.export.ServingInputReceiver(features, json)
```

Output shape:
[batch, width, height, 3]

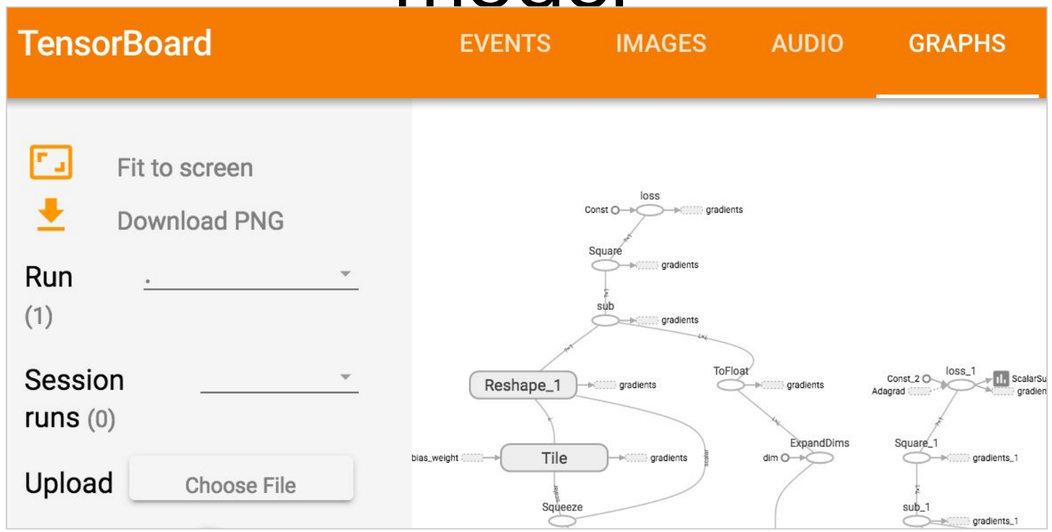
# Example serving input function that decodes JPEGs

JSON

```
{"jpeg_bytes":{"b64":"/9j/4DAwQDBAgEBAgQCwkLEBAQEBAQEQEB...”}}
```

Base 64: special syntax _bytes and b64

# Lab: Implementing a distributed TensorFlow model




- Use train_and_evaluate
- Monitor training using TensorBoard

| | Problem | Solution |
|---|---|---|
| ✓ | Out of memory data | Use the Dataset API |
| ✓ | Distribution | Use `train_and_evaluate` |
| ✓ | Need to evaluate during training | Use `train_and_evaluate` + TensorBoard |
| ✓ | Deployments that scale | Use serving input function |

cloud.google.com