Google Cloud

Custom Estimator

Lak Lakshmanan

Machine Learning on
Google Cloud Platform

The Art of ML

Hyperparameter Tuning

A Pinch of Science

The Science of Neural Networks

Embeddings

Custom Estimator

Learn how to...

Go beyond canned estimators

Learn how to...

Go beyond canned estimators

Write a custom estimator

Learn how to...

Go beyond canned estimators

Write a custom estimator

Gain control over model functions

Learn how to...

Go beyond canned estimators

Write a custom estimator

Gain control over model functions

Incorporate Keras models into Estimator

Estimator provides a lot of benefits

- 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

Suppose that you want to use a model structure from a research paper ...

```
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

def input_fn():
   return tf.data.Dataset.list_files("train.csv-*")    \
                      .flat_map(tf.data.TextLineDataset) \
                      .map(decode_line).shuffle(1000) \
                      .repeat(15).batch(128).
                      .make_one_shot_iterator().get_next()

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

```
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_receive
r_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)
```

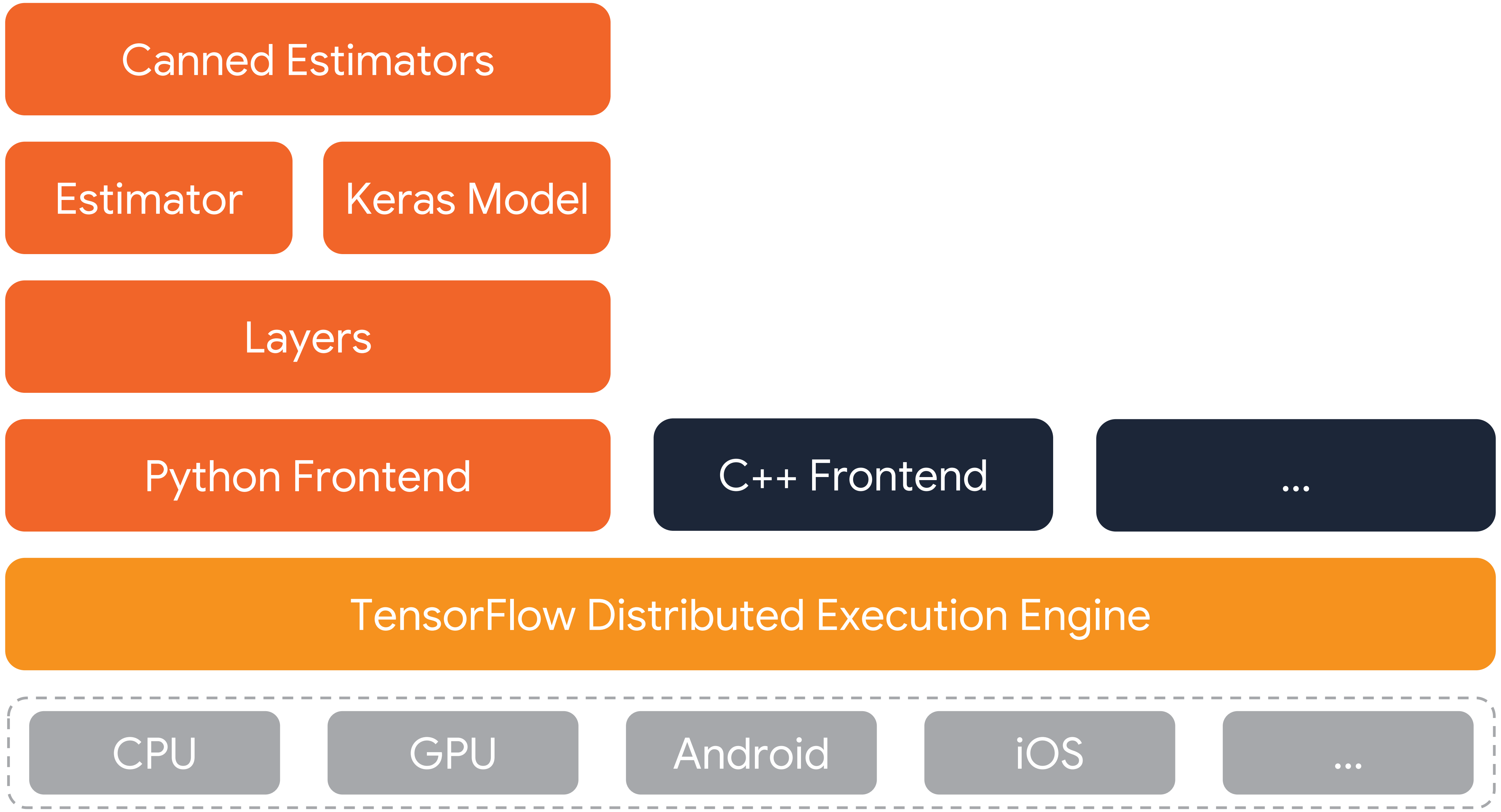
```
run_config = tf.estimator.RunConfig(model_dir=output_dir, ...)

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)
```

```
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_receive
r_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)
```

```
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_receive
r_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)
```

```
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_receive
r_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)
```

Canned Estimators are sometimes insufficient



Suppose that you want to use a model structure from a research paper ...

Implement the model using low-level TensorFlow ops

```
def model_from_research_paper(timeseries):
  x = tf.split(timeseries, N_INPUTS, 1)
  lstm_cell = rnn.BasicLSTMCell(LSTM_SIZE, forget_bias=1.0)
  outputs, _ = rnn.static_rnn(lstm_cell, x, dtype=tf.float32)
  outputs = outputs[-1]
  weight = tf.Variable(tf.random_normal([LSTM_SIZE, N_OUTPUTS]))
  bias = tf.Variable(tf.random_normal([N_OUTPUTS]))
  predictions = tf.matmul(outputs, weight) + bias
  return predictions
```

How do we wrap this custom model into Estimator framework?

Create train_and_evaluate function with the base-class Estimator

```
def train_and_evaluate(output_dir, ...):
  estimator = tf.estimators.Estimator(model_fn = myfunc,
    model_dir = output_dir),
  train_spec = get_train()
  exporter = ...
  eval_spec = get_valid()
  tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

myfunc (above) is a EstimatorSpec

The 6 things in a EstimatorSpec

```
def myfunc(features, targets, mode):
  # Code up the model
  predictions = model_from_research_paper(features[INCOL])

  # Set up loss function, training/eval ops
  … #(next slide)

  # Create dictionary of output tensors
  predictions_dict = {"predicted": predictions}

  # Create export outputs
  export_outputs = {"regression_export_outputs":
    tf.estimator.export.RegressionOutput(value = predictions)}

  # Return EstimatorSpec
  return tf.estimator.EstimatorSpec(
      mode = mode,
      predictions = predictions_dict,
      loss = loss,
      train_op = train_op,
      eval_metric_ops = eval_metric_ops,
      export_outputs = export_outputs)
```

1. Mode is pass-through
2. Any tensors you want to return
3. Loss metric
4. Training op
5. Eval ops
6. Export outputs

The ops are set up in the appropriate mode

```
if mode == tf.estimator.ModeKeys.TRAIN or
   mode == tf.estimator.ModeKeys.EVAL:
     loss = tf.losses.mean_squared_error(targets, predictions)
     train_op = tf.contrib.layers.optimize_loss(
         loss=loss,
         global_step=tf.contrib.framework.get_global_step(),
         learning_rate=0.01,
         optimizer="SGD")
     eval_metric_ops = {
      "rmse": tf.metrics.root_mean_squared_error(targets,
predictions)
     }
else:
     loss = None
     train_op = None
     eval_metric_ops = None
```

Create train_and_evaluate function with the base-class Estimator

```
def train_and_evaluate(output_dir, ...):
  estimator = tf.estimators.Estimator(model_fn = myfunc,
     model_dir = output_dir),
  train_spec = get_train()
  exporter = ...
  eval_spec = get_valid()
  tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

myfunc (above) is an EstimatorSpec

Lab

Implementing a Custom Estimator

Lak Lakshmanan

Lab: Implementing a Custom Estimator

You will build a custom estimator to predict time series values

Create train_and_evaluate function with the base-class Estimator

```
def train_and_evaluate(output_dir, ...):
  estimator = tf.estimators.Estimator(model_fn = myfunc,
    model_dir = output_dir),
  train_spec = get_train()
  exporter = ...
  eval_spec = get_valid()
  tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

myfunc (above) is an EstimatorSpec; how does it work with Keras?

Keras is a high-level deep neural networks library that supports multiple backends

Keras

TensorFlow

theano

Microsoft CNTK

Keras is easy to use for fast prototyping

```
model = Sequential()
model.add(Embedding(max_features, output_dim=256))
model.add(LSTM(128))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train, batch_size=16, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=16)
```

From a compiled Keras model, you can get an Estimator

```
from tensorflow import keras

model = Sequential()
model.add(Embedding(max_features, output_dim=256))
model.add(LSTM(128))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])


estimator = keras.estimator.model_to_estimator(keras_model=model)
```

~~model.fit(x_train, y_train, batch_size=16, epochs=10)~~
~~score = model.evaluate(x_test, y_test, batch_size=16)~~

From a compiled Keras model, you can get an Estimator

```
def train_and_evaluate(output_dir):
    estimator = make_keras_estimator(output_dir)
    train_spec = tf.estimator.TrainSpec(train_fn,
                                        max_steps = 1000)
    exporter = LatestExporter('exporter', serving_input_fn)
    eval_spec = tf.estimator.EvalSpec(eval_fn,
                                      steps = None,
                                      exporters = exporter)
    tf.estimator.train_and_evaluate(estimator, train_spec,
eval_spec)
```

The connection between the input features and Keras is through a naming convention

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(..., name='XYZ'))


def train_input_fn():
    ...
    features = {
                  'XYZ_input': some_tensor,
               }
    return features, labels
```

cloud.google.com