TensorFlow and Keras

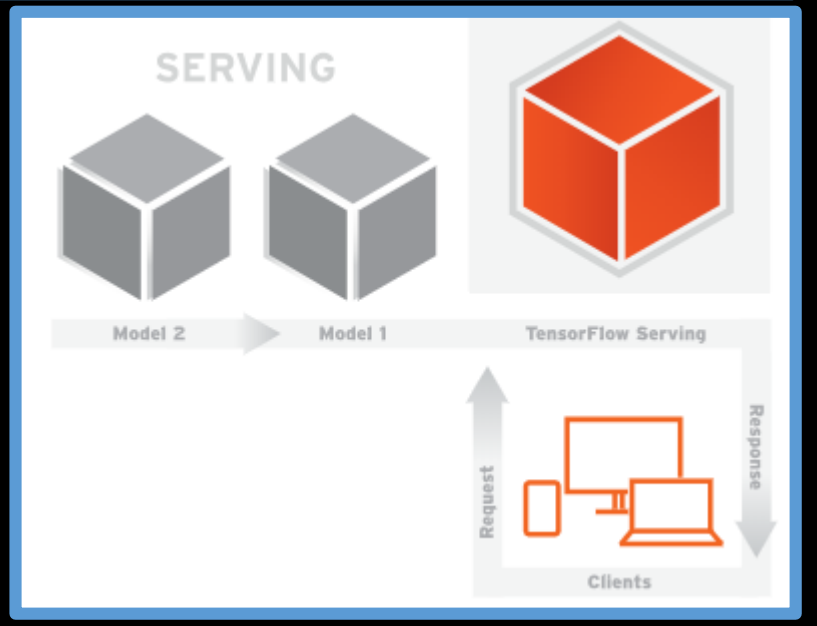
Palacode Narayana Iyer Anantharaman

9 Aug 2018

TensorFlow products, variants

- Core products and variants
  - TensorFlow 1.9 (CPU, GPU, TPU)
  - TensorFlow js
  - TensorFlow lite

- Deployment and Serving
  - TensorFlow Serving



- Development Tools: Debug, Visualize
  - TensorFlow Debugger
  - Tensorboard visualizer

What is TensorFlow (TF)?

TensorFlow is an open-source software library for dataflow programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks.[3] It is used for both research and production at Google,[3]:min 0:15/2:17 [4]:p.2 [3]:0:26/2:17 often replacing its closed-source predecessor, DistBelief.

- TF is a framework with the following characteristics:
  - A Framework that allows a developer to express his machine learning algorithm symbolically, performing compilation of these statements and executing them.
  - A programming metaphor that requires the developer to model the machine learning algorithm as a computation graph.
  - A set of Python classes and methods that provide an API interface
  - A re-targetable system that can run on different hardware

Why use TF?

- Suppose we are addressing a problem using a machine learning approach and our proposed solution can be implemented by one of the standard classifier architectures. (say, an SVM)
- We often use one of the standard ML libraries (say, scikit) for the above situation instead of implementing our own classifier
- If the existing ML models aren't optimally suited for our problem, we might come up with a custom architecture that might give a higher accuracy. In such a case we might end up building the classifier from scratch, starting with mathematical formulations and implementing ground up.
- Problems that call for deep learning oriented solutions require custom architectures.
- A lego-piece based approach where a library offers basic building blocks would be more suited. With in-built support to run on GPUs, such libraries are well suited for custom designs
- With TF, you can quickly build arbitrarily complex architectures

Keras

- Evolved as an abstraction layer that abstracts Theano, TensorFlow, etc
- The API's are high level and enables faster development
- Keras is now integrated with TensorFlow.
- Thus, one can develop a pure Keras application or can use it as a part of a TF code.

Guiding principles

- **User friendliness.** Keras is an API designed for human beings, not machines. It puts user experience front and center. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.
- **Modularity.** A model is understood as a sequence or a graph of standalone, fully-configurable modules that can be plugged together with as few restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions, regularization schemes are all standalone modules that you can combine to create new models.
- **Easy extensibility.** New modules are simple to add (as new classes and functions), and existing modules provide ample examples. To be able to easily create new modules allows for total expressiveness, making Keras suitable for advanced research.
- **Work with Python.** No separate models configuration files in a declarative format. Models are described in Python code, which is compact, easier to debug, and allows for ease of extensibility.

Keras and TF: Which to choose?

- Standard ML models (Such as a linear regression) can be implemented with Keras with just a few lines of code.

- To implement the same with TF took not only many more lines of code but also require an understanding of the TF metaphor (such as: Compute Graph, Sessions, Name spaces etc)
  - This is particularly true of earlier versions of TF but are getting easier with versions post 1.7

- Given a problem on hand, how to choose the framework we need?

Choosing between Keras/TF/Both

- There can be different ways to decide on this, the following is my thinking:
  - Even with TF 1.9, a pure Keras application is easier/faster to develop. If your application can be implemented in pure Keras, we can choose it
  - Keras, now being part of Google, can now be expected to be more focused on TensorFlow and might have less support for other frameworks in future (just a guess!)
  - The core development tools such as debugger, visualization, deployment models etc would continue to get strengthened on TF product lines, apart from variants for Javascript and mobile.
  - Since Keras is fully integrated with TF, it's a good idea to choose TF as our primary framework and implement the complete models with TF

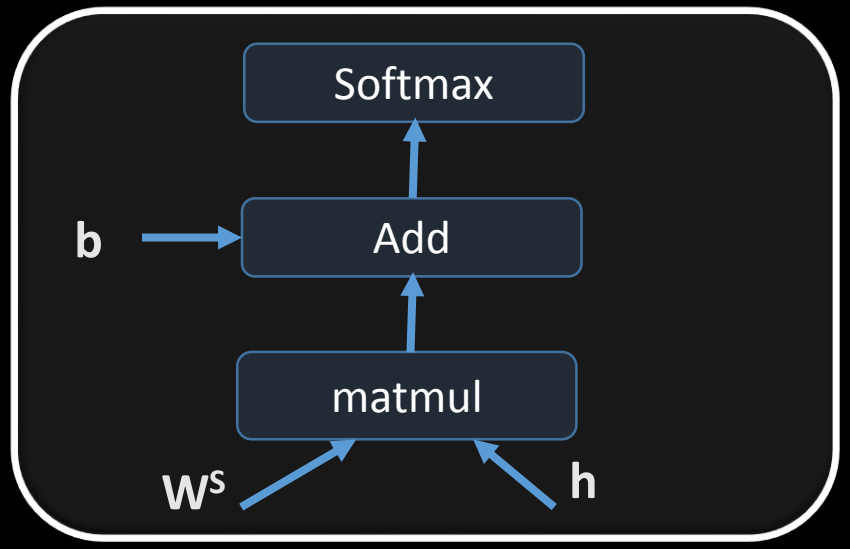
Key Topics

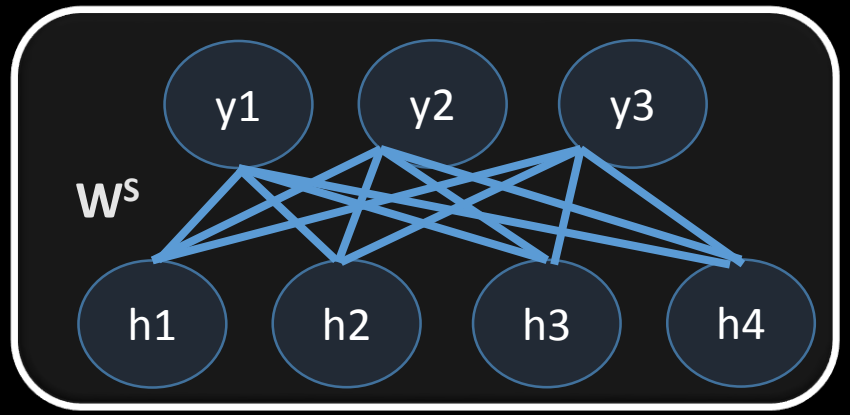
- Low level APIs and TF metaphor
- High Level APIs
- Estimators
- TF Serving
- TF Debugger
- TensorBoard

Training an ML model

- A machine learning model (such as a neural network) can be represented as a graph
  - Neurons (input, hidden, output) are the nodes of the graph and the parameters (weights, bias terms) are the edges
- We train the model using backpropagation, computing the derivatives for each learnable parameter, starting from the output layer
- Traditional approach: Design the architecture, derive expressions for derivatives, implement the architecture
  - Hand calculation of these derivatives and implementation is error prone and time consuming for non trivial networks
- TensorFlow allows us to specify the graph and performs auto differentiation
- The key idea is to view the model as a composition of layers, implementing the forward and backward propagations exclusively for each layer (See the reference)

TF Compute Graph



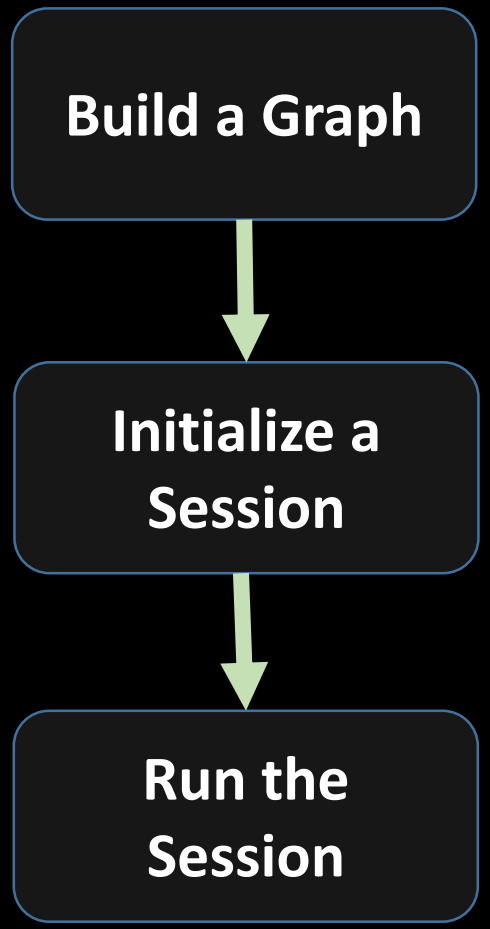


- Operations are the nodes of the graph
- Tensors constitute the edges

$$P(y|h) = Softmax(dot\_prod)$$

$$dot\_prod = (W^S h + b)$$

Developing with TF: High level steps



- Perform any pre-processing steps to set up the dataset for the classifier.
- Set up the variables (such as the Weights and biases) and placeholders (such as the input vector) that are fed at runtime
- Define the required computations. E.g. you may define the dot_product computation using the matmul method of tensorflow feeding it the weight matrix variable and the input vector (that may be defined as a placeholder).

  Note: Till this point nothing is executed by Tensorflow
- Start a session instance and initialize all variables
- Run the session

Some methods to get started

- tf.constant()
- tf.placeholder()
- tf.variable()
- tf.matmul( )
- tf.softmax( )
- tf.reduce_sum( )
- tf.train.GradientDescentOptimizer( )
- tf.initialize_all_variables( )
- tf.Session( )
- tf.run( )

Example 1

```
In [1]: import tensorflow as tf

In [2]: import numpy as np

In [3]: c1 = tf.constant([1, 2, 3, 4])

In [4]: c2 = tf.constant([-1, 2, -3, 4])

In [5]: y = c1 + c2 #symbolic addition

In [8]: # y is now a place holder that will receive the value of c1 + c2 once we run this...currently it is just another tensor

In [9]: print y
        Tensor("add:0", shape=(4,), dtype=int32)

In [10]: sess = tf.Session()

In [13]: y = sess.run(y)

In [14]: print y
         [0 4 0 8]
```

Example 2

We will illustrate comptation of Wx + b in this demo

In [13]:
```
x = tf.placeholder(tf.float64, (10, 1)) # x will hold an input of 10 dims - we will feed the input later
```

In [14]:
```
W = tf.Variable(np.random.randn(7, 10)*0.01) # assume our hidden size = 7 and input size = 10
```

In [23]:
```
b = tf.Variable(tf.zeros([7], dtype=tf.float64))
```

In [24]:
```
h = tf.matmul(W, x) + b
```

In [25]:
```
sess = tf.Session()
```

In [34]:
```
sess.run(tf.initialize_all_variables())
```

In [35]:
```
h = sess.run(h, feed_dict={x: [[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]]})
```

In [36]:
```
print h
```

```
[[ 0.39385701  0.39385701  0.39385701  0.39385701  0.39385701  0.39385701
   0.39385701]
 [-0.15568769 -0.15568769 -0.15568769 -0.15568769 -0.15568769 -0.15568769
  -0.15568769]
 [ 0.07021094  0.07021094  0.07021094  0.07021094  0.07021094  0.07021094
   0.07021094]
 [-0.03489276 -0.03489276 -0.03489276 -0.03489276 -0.03489276 -0.03489276
  -0.03489276]
 [ 0.2863619   0.2863619   0.2863619   0.2863619   0.2863619   0.2863619
```

Life cycle of TensorFlow node values

- See the explanation on the board

High Level APIs

- Keras layer integrated in to TF
- Eager Execution
- Estimators
- Data Pipelines

High Level APIs: keras

- tf.keras is TensorFlow's implementation of the Keras API specification.
- This is a high-level API to build and train models that includes first-class support for TensorFlow-specific functionality, such as eager execution, tf.data pipelines, and Estimators.
- tf.keras makes TensorFlow easier to use without sacrificing flexibility and performance.

TF Keras Example

```
import tensorflow as tf
from tensorflow import keras
model = keras.Sequential()
# Adds a densely-connected layer with 64 units to the model:
model.add(keras.layers.Dense(64, activation='relu'))
# Add another:
model.add(keras.layers.Dense(64, activation='relu'))
# Add a softmax layer with 10 output units:
model.add(keras.layers.Dense(10, activation='softmax'))
model.compile(optimizer=tf.train.AdamOptimizer(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Data Pipeline

- For small amount of data (say 10K samples, 10 features), we can keep the complete data in memory and use fit() method as below:

```
import numpy as np
data = np.random.random((1000, 32))
labels = np.random.random((1000, 10))
val_data = np.random.random((100, 32))
val_labels = np.random.random((100, 10))
model.fit(data, labels, epochs=10, batch_size=32,
        validation_data=(val_data, val_labels))
```

- Use the Datasets API to scale to large datasets or multi-device training. Pass a tf.data.Dataset instance to the fit method:

Datasets API

```
dataset = tf.data.Dataset.from_tensor_slices((data, labels))
dataset = dataset.batch(32).repeat()

val_dataset = tf.data.Dataset.from_tensor_slices((val_data, val_labels))
val_dataset = val_dataset.batch(32).repeat()

model.fit(dataset, epochs=10, steps_per_epoch=30,
          validation_data=val_dataset,
          validation_steps=3)
```

Evaluate and Predict

```
model.evaluate(x, y, batch_size=32)
model.evaluate(dataset, steps=30)
```

```
model.predict(x, batch_size=32)
model.predict(dataset, steps=30)
```

Keras Functional API

- tf.keras.Sequential is useful for implementing a stack of layers
- When the model is more complex and arbitrary, use functional API

- Multi-input models,
- Multi-output models,
- Models with shared layers (the same layer called several times),
- Models with non-sequential data flows (e.g. residual connections).

Building a model with the functional API works like this:

1. A layer instance is callable and returns a tensor.
2. Input tensors and output tensors are used to define a `tf.keras.Model` instance.
3. This model is trained just like the `Sequential` model.

Example

```
inputs = keras.Input(shape=(32,))  # Returns a placeholder tensor
# A layer instance is callable on a tensor, and returns a tensor.
x = keras.layers.Dense(64, activation='relu')(inputs)
x = keras.layers.Dense(64, activation='relu')(x)
predictions = keras.layers.Dense(10, activation='softmax')(x)
# Instantiate the model given inputs and outputs.
model = keras.Model(inputs=inputs, outputs=predictions)
# The compile step specifies the training configuration.
model.compile(optimizer=tf.train.RMSPropOptimizer(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
# Trains for 5 epochs
model.fit(data, labels, batch_size=32, epochs=5)
```

Keras Custom Layers

- You can create a custom layer by sub classing tf.keras.layers.layer

- Implement the following methods:

  - `build` : Create the weights of the layer. Add weights with the `add_weight` method.
  - `call` : Define the forward pass.
  - `compute_output_shape` : Specify how to compute the output shape of the layer given the input shape.
  - Optionally, a layer can be serialized by implementing the `get_config` method and the `from_config` class method.

Custom Layer Example

```
class MyLayer(keras.layers.Layer):

  def __init__(self, output_dim, **kwargs):
    self.output_dim = output_dim
    super(MyLayer, self).__init__(**kwargs)

  def build(self, input_shape):
    shape = tf.TensorShape((input_shape[1], self.output_dim))
    # Create a trainable weight variable for this layer.
    self.kernel = self.add_weight(name='kernel',
                                  shape=shape,
                                  initializer='uniform',
                                  trainable=True)
    # Be sure to call this at the end
    super(MyLayer, self).build(input_shape)

  def call(self, inputs):
    return tf.matmul(inputs, self.kernel)
```

```
def compute_output_shape(self, input_shape):
  shape = tf.TensorShape(input_shape).as_list()
  shape[-1] = self.output_dim
  return tf.TensorShape(shape)

def get_config(self):
  base_config = super(MyLayer, self).get_config()
  base_config['output_dim'] = self.output_dim

@classmethod
def from_config(cls, config):
  return cls(**config)
```

```
# Create a model using the custom layer
model = keras.Sequential([MyLayer(10),
                          keras.layers.Activation('softmax')])

# The compile step specifies the training configuration
model.compile(optimizer=tf.train.RMSPropOptimizer(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Trains for 5 epochs.
model.fit(data, targets, batch_size=32, epochs=5)
```

Callbacks

Callbacks

A callback is an object passed to a model to customize and extend its behavior during training. You can write your own custom callback, or use the built-in `tf.keras.callbacks` that include:

- `tf.keras.callbacks.ModelCheckpoint`: Save checkpoints of your model at regular intervals.
- `tf.keras.callbacks.LearningRateScheduler`: Dynamically change the learning rate.
- `tf.keras.callbacks.EarlyStopping`: Interrupt training when validation performance has stopped improving.
- `tf.keras.callbacks.TensorBoard`: Monitor the model's behavior using TensorBoard.

To use a `tf.keras.callbacks.Callback`, pass it to the model's `fit` method:

Example

```
callbacks = [
  # Interrupt training if `val_loss` stops improving for over 2 epochs
  keras.callbacks.EarlyStopping(patience=2, monitor='val_loss'),
  # Write TensorBoard logs to `./logs` directory
  keras.callbacks.TensorBoard(log_dir='./logs')
]
model.fit(data, labels, batch_size=32, epochs=5, callbacks=callbacks,
          validation_data=(val_data, val_targets))
```

Code Walkthrough

Saving/Restoring the model

- Save and Restore model weights
- Save and restore model configuration (JSON, YAML serialization)
- Save and Restore the entire model
  - Saves weight values, the model's configuration, optimizer's configuration
  - Allows checkpointing the model to resume later from exactly the same state even without the original source code

```
# Save entire model to a HDF5 file
model.save('my_model.h5')

# Recreate the exact same model, including weights and optimizer
model = keras.models.load_model('my_model.h5')
```

Eager Execution

- The TF compute graph metaphor where the model is specified symbolically requires compilation and run time binding to actual inputs.
- Often it is a bit confusing for those who expect the statements to be “interpreted” instantly as in Python
- Eager Execution allows instant execution without the need for starting a session, binding variables with feed_dict and so on
- This helps debugging as it provides instant feedback and also is more intuitive interface for a python developer as this is more “pythonic”

Sample Code

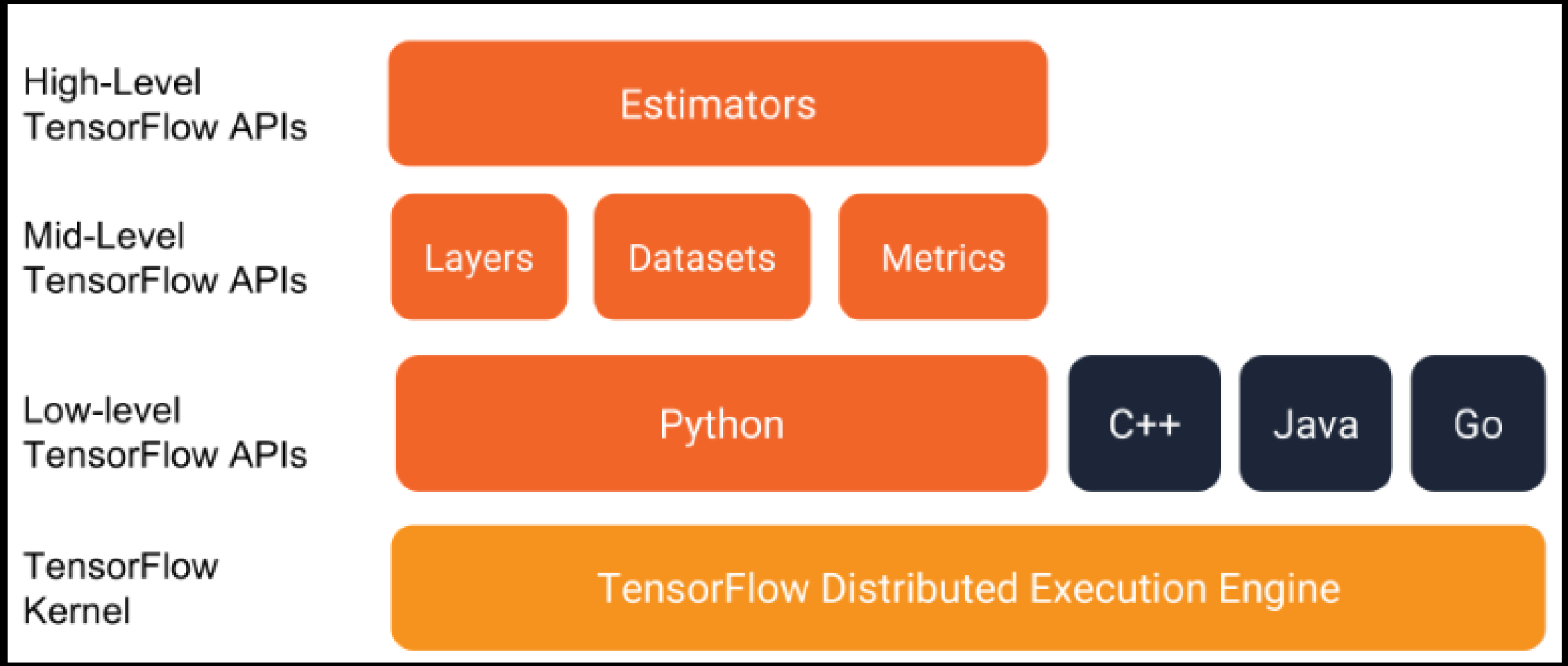
```
from __future__ import absolute_import, division, print_function
import tensorflow as tf

tf.enable_eager_execution()

tf.executing_eagerly()        # => True

x = [[2.]]

m = tf.matmul(x, x)

print("hello, {}".format(m))  # => "hello, [[4.]]"
```

Estimators

Estimators are high level APIs that have the following advantages:

- Estimators are easier to program as they are high level API
- They support distributed environment, on CPU/GPU/TPU without recoding
- Estimators make graph building transparent
- A number of pre-made estimators are available

Estimators



Pre-made estimators

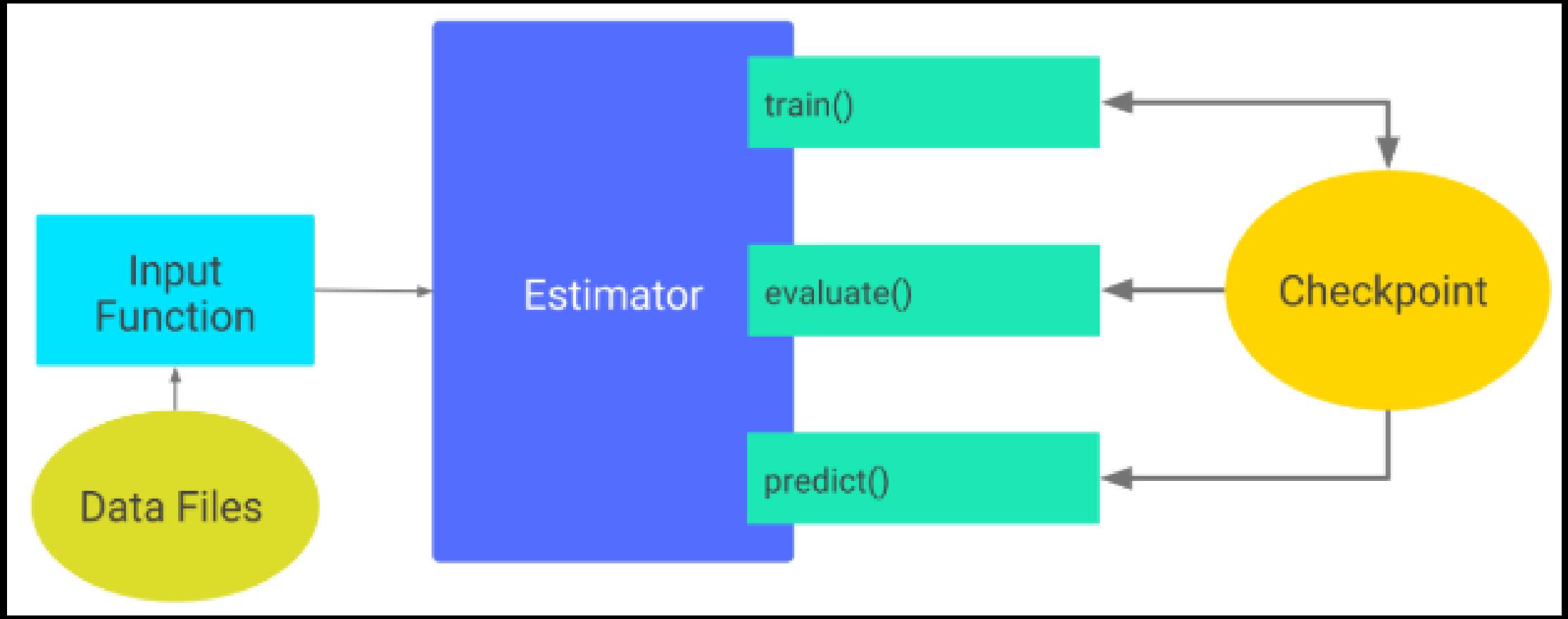
Pre-made Estimators

Pre-made Estimators enable you to work at a much higher conceptual level than the base TensorFlow APIs. You no longer have to worry about creating the computational graph or sessions since Estimators handle all the "plumbing" for you. That is, pre-made Estimators create and manage `Graph` and `Session` objects for you. Furthermore, pre-made Estimators let you experiment with different model architectures by making only minimal code changes. `DNNClassifier`, for example, is a pre-made Estimator class that trains classification models based on dense, feed-forward neural networks.

Four Steps to a pre-made estimator application

1. Write one or more dataset importing functions
2. Define the feature columns
3. Instantiate the required pre-made estimator
4. Call a training, evaluation, predict function

Estimators and Checkpointing



Sample Code

```
def input_fn(dataset):
  ...  # manipulate dataset, extracting the feature dict and the label
  return feature_dict, label
# Define three numeric feature columns.
population = tf.feature_column.numeric_column('population')
crime_rate = tf.feature_column.numeric_column('crime_rate')
median_education = tf.feature_column.numeric_column('median_education',
                   normalizer_fn=lambda x: x - global_education_mean)
# Instantiate an estimator, passing the feature columns.
estimator = tf.estimator.LinearClassifier(feature_columns=[population, crime_rate,
median_education],)
# my_training_set is the function created in Step 1
estimator.train(input_fn=my_training_set, steps=2000)
```

Recommended Workflow

Recommended workflow

We recommend the following workflow:

1. Assuming a suitable pre-made Estimator exists, use it to build your first model and use its results to establish a baseline.
2. Build and test your overall pipeline, including the integrity and reliability of your data with this pre-made Estimator.
3. If suitable alternative pre-made Estimators are available, run experiments to determine which pre-made Estimator produces the best results.
4. Possibly, further improve your model by building your own custom Estimator.

Keras Model to Estimator

```
model = keras.Sequential([layers.Dense(10,activation='softmax'),
                          layers.Dense(10,activation='softmax')])


model.compile(optimizer=tf.train.RMSPropOptimizer(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])


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

Creating Estimators From Keras Models

```
keras_inception_v3 = tf.keras.applications.inception_v3.InceptionV3(weights=None) # # Instantiate a Keras inception v3 model.

# Compile model with the optimizer, loss, and metrics you'd like to train with.

keras_inception_v3.compile(optimizer=tf.keras.optimizers.SGD(lr=0.0001, momentum=0.9),  loss='categorical_crossentropy',
metric='accuracy')

# Create an Estimator from the compiled Keras model. The initial model state of the keras model is preserved in the created Estimator.

est_inception_v3 = tf.keras.estimator.model_to_estimator(keras_model=keras_inception_v3)

# Treat the derived Estimator as you would with any other Estimator.

# First, recover the input name(s) of Keras model, so we can use them as the feature column name(s) of the Estimator input function:

keras_inception_v3.input_names  # print out: ['input_1']

# Once we have the input name(s), we can create the input function, for example,

# for input(s) in the format of numpy ndarray:

train_input_fn = tf.estimator.inputs.numpy_input_fn(

    x={"input_1": train_data},

    y=train_labels,

    num_epochs=1,

    shuffle=False)

# To train, we call Estimator's train function:

est_inception_v3.train(input_fn=train_input_fn, steps=2000)
```

Lab Assignment

- You will be provided with the dataset for classification and regression
- You are required to use the linear model estimators to perform the classification and regression tasks
- You are required to identify the formula f(x) and guess what the input and output quantities represent ☺

For details refer the slide deck dated 14 Aug 2018 on lab assignment

TensorFlow and Keras
Part 2

Palacode Narayana Iyer Anantharaman

16 Aug 2018

Typical ML pipeline

- Most of the time in building a ML system is spent on forming the dataset and manipulating it
- Datasets are often created from one or more files. Special cases are the data that are generated at run time e.g. data streams captured from live websites
- Files may be: Unformatted Text, CSV text, JSON, Binary records etc
- Often a subset of fields may be used as features or labels
- Need for preprocessing before being fed to the model
- TensorFlow allows a consistent and reusable way to handle this task

Input Pipelines

- Datasets used for deep learning are huge
- It is often necessary to build a complex pipeline to feed the data to the model
- Data may arise from distributed sources
- Data from these may need to be augmented and merged to form a batch for training
- TF Data API supports such complex use cases
- These can be easily integrated with estimators

	Train	Validation	Test	# Classes	# Trainable Classes
Images	9,011,219	41,620	125,436	-	-
Machine-Generated Labels	78,977,695	512,093	1,545,835	7,870	4,764
Human-Verified Labels	27,894,289 pos: 13,444,569 neg: 14,449,720	551,390 pos: 365,772 neg: 185,618	1,667,399 pos: 1,105,052 neg: 562,347	19,794	7,186

The COCO train, validation, and test sets, containing more than 200,000 images and 80 object categories, are available on the download page. All object instances are annotated with a detailed segmentation mask. Annotations on the training and validation sets (with over 500,000 object instances segmented) are publicly available.

TF Data

The `tf.data` API introduces two new abstractions to TensorFlow:

- A `tf.data.Dataset` represents a sequence of elements, in which each element contains one or more `Tensor` objects. For example, in an image pipeline, an element might be a single training example, with a pair of tensors representing the image data and a label. There are two distinct ways to create a dataset:
  - Creating a **source** (e.g. `Dataset.from_tensor_slices()`) constructs a dataset from one or more `tf.Tensor` objects.
  - Applying a **transformation** (e.g. `Dataset.batch()`) constructs a dataset from one or more `tf.data.Dataset` objects.
- A `tf.data.Iterator` provides the main way to extract elements from a dataset. The operation returned by `Iterator.get_next()` yields the next element of a `Dataset` when executed, and typically acts as the interface between input pipeline code and your model. The simplest iterator is a "one-shot iterator", which is associated with a particular `Dataset` and iterates through it once. For more sophisticated uses, the `Iterator.initializer` operation enables you to reinitialize and parameterize an iterator with different datasets, so that you can, for example, iterate over training and validation data multiple times in the same program.

Dataset Input Pipeline: Readers

Creating new datasets

Static methods in `Dataset` that create new datasets.

- `tf.data.Dataset.from_generator`
- `tf.data.Dataset.from_tensor_slices`
- `tf.data.Dataset.from_tensors`
- `tf.data.Dataset.list_files`
- `tf.data.Dataset.range`
- `tf.data.Dataset.zip`

Reader classes

Classes that create a dataset from input files.

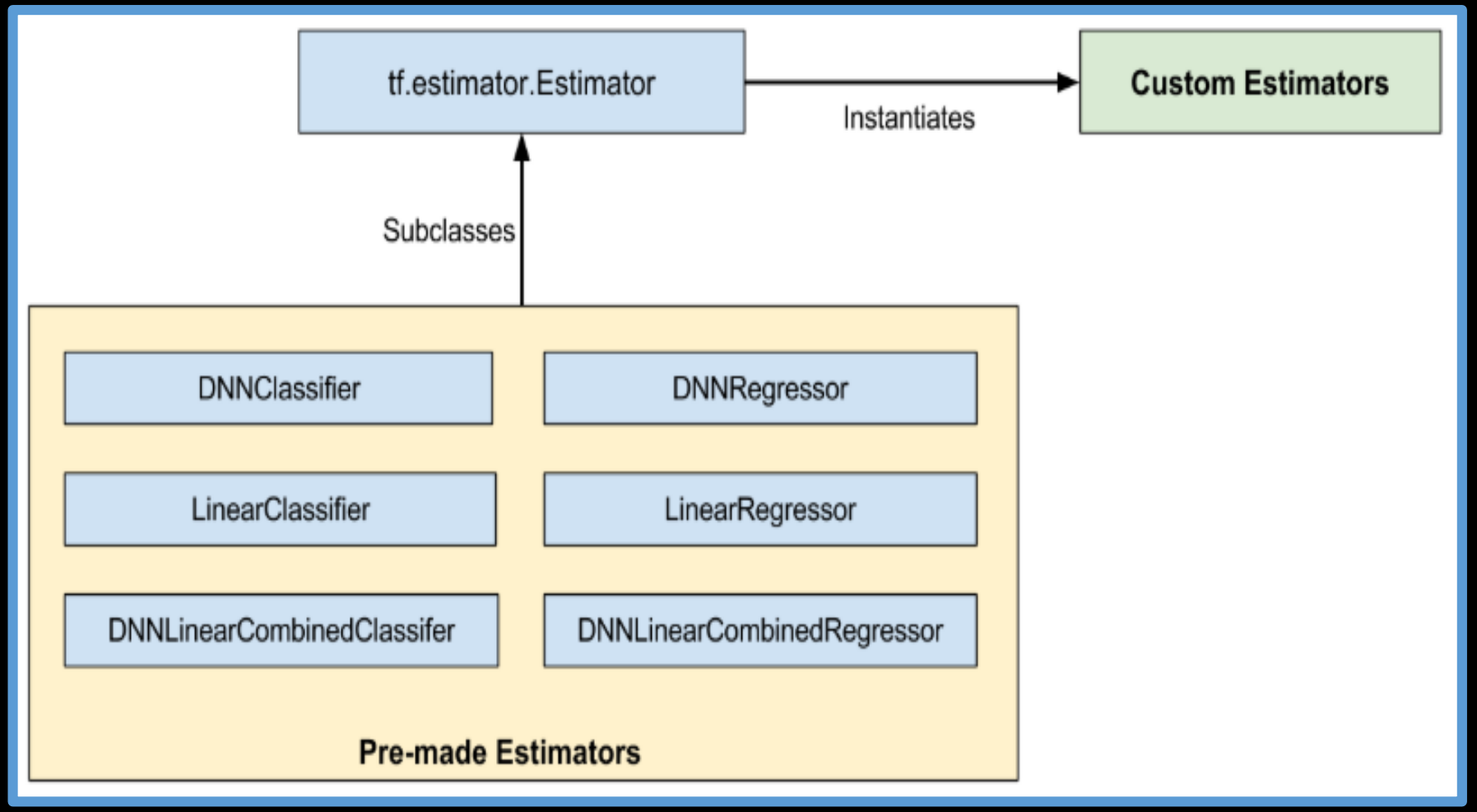
- `tf.data.FixedLengthRecordDataset`
- `tf.data.TextLineDataset`
- `tf.data.TFRecordDataset`

```
train_data = train_data.batch(100).shuffle().repeat()
```

- `tf.data.Dataset.apply`
- `tf.data.Dataset.batch`
- `tf.data.Dataset.cache`
- `tf.data.Dataset.concatenate`
- `tf.data.Dataset.filter`
- `tf.data.Dataset.flat_map`
- `tf.data.Dataset.interleave`
- `tf.data.Dataset.map`
- `tf.data.Dataset.padded_batch`
- `tf.data.Dataset.prefetch`
- `tf.data.Dataset.repeat`
- `tf.data.Dataset.shard`
- `tf.data.Dataset.shuffle`
- `tf.data.Dataset.skip`
- `tf.data.Dataset.take`

Pre-made and custom estimators

- Pre made estimators are sub classes of Estimator. They are fully “baked”
- We may build a custom Estimator when:
  - (a) We don’t find a ready to use pre made estimator
  - (b) We need some custom added functionality (e.g: a custom metric to be computed)



TensorBoard