TensorFlow 2.0 Beta is available **Learn more** (/beta/)

Keras

Run in Google (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/er Colab

Keras is a high-level API to build and train deep learning models. It's used for fast prototyping, advanced research, and production, with three key advantages:

- *User friendly*
  Keras has a simple, consistent interface optimized for common use cases. It provides clear and actionable feedback for user errors.
- *Modular and composable*
  Keras models are made by connecting configurable building blocks together, with few restrictions.
- *Easy to extend*
  Write custom building blocks to express new ideas for research. Create new layers, loss functions, and develop state-of-the-art models.

Import tf.keras

`tf.keras` (https://www.tensorflow.org/api_docs/python/tf/keras) is TensorFlow's implementation of the Keras API specification (https://keras.io). This is a high-level API to build and train models that includes first-class support for TensorFlow-specific functionality, such as eager execution (#eager_execution), `tf.data` (https://www.tensorflow.org/api_docs/python/tf/data) pipelines, and Estimators (https://www.tensorflow.org/guide/estimators). `tf.keras` (https://www.tensorflow.org/api_docs/python/tf/keras) makes TensorFlow easier to use without sacrificing flexibility and performance.

To get started, import **`tf.keras`** (https://www.tensorflow.org/api_docs/python/tf/keras) as part of your TensorFlow program setup:

```
!pip install -q pyyaml  # Required to save models in YAML format
```

```
from __future__ import absolute_import, division, print_function, unicode_litera

import tensorflow as tf
from tensorflow.keras import layers

print(tf.version.VERSION)
print(tf.keras.__version__)
```

```
1.14.0
2.2.4-tf
```

`tf.keras` (https://www.tensorflow.org/api_docs/python/tf/keras) can run any Keras-compatible code, but keep in mind:

- The **`tf.keras`** (https://www.tensorflow.org/api_docs/python/tf/keras) version in the latest TensorFlow release might not be the same as the latest `keras` version from PyPI. Check `tf.keras.__version__`.
- When saving a model's weights (#weights_only), **`tf.keras`** (https://www.tensorflow.org/api_docs/python/tf/keras) defaults to the checkpoint format (https://www.tensorflow.org/guide/checkpoints). Pass `save_format='h5'` to use HDF5.

Build a simple model

Sequential model

In Keras, you assemble *layers* to build *models*. A model is (usually) a graph of layers. The most common type of model is a stack of layers: the **`tf.keras.Sequential`** (https://www.tensorflow.org/api_docs/python/tf/keras/Sequential) model.

To build a simple, fully-connected network (i.e. multi-layer perceptron):

```
model = tf.keras.Sequential()
# Adds a densely-connected layer with 64 units to the model:
model.add(layers.Dense(64, activation='relu'))
# Add another:
model.add(layers.Dense(64, activation='relu'))
# Add a softmax layer with 10 output units:
model.add(layers.Dense(10, activation='softmax'))
```

```
WARNING: Logging before flag parsing goes to stderr.
W0708 23:41:18.068892 140374570354432 deprecation.py:506] From /tmpfs/src/tf_doc
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the c
```

Configure the layers

There are many **`tf.keras.layers`** (https://www.tensorflow.org/api_docs/python/tf/keras/layers) available with some common constructor parameters:

- **`activation`**: Set the activation function for the layer. This parameter is specified by the name of a built-in function or as a callable object. By default, no activation is applied.
- **`kernel_initializer`** and **`bias_initializer`**: The initialization schemes that create the layer's weights (kernel and bias). This parameter is a name or a callable object. This defaults to the "**`Glorot uniform`**" initializer.
- **`kernel_regularizer`** and **`bias_regularizer`**: The regularization schemes that apply the layer's weights (kernel and bias), such as L1 or L2 regularization. By default, no regularization is applied.

The following instantiates **`tf.keras.layers.Dense`** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense) layers using constructor arguments:

```
# Create a sigmoid layer:
layers.Dense(64, activation='sigmoid')
# Or:
layers.Dense(64, activation=tf.sigmoid)

# A linear layer with L1 regularization of factor 0.01 applied to the kernel mat
layers.Dense(64, kernel_regularizer=tf.keras.regularizers.l1(0.01))

# A linear layer with L2 regularization of factor 0.01 applied to the bias vecto
layers.Dense(64, bias_regularizer=tf.keras.regularizers.l2(0.01))

# A linear layer with a kernel initialized to a random orthogonal matrix:
layers.Dense(64, kernel_initializer='orthogonal')

# A linear layer with a bias vector initialized to 2.0s:
layers.Dense(64, bias_initializer=tf.keras.initializers.constant(2.0))
```

```
<tensorflow.python.keras.layers.core.Dense at 0x7faafa157208>
```

Train and evaluate

Set up training

After the model is constructed, configure its learning process by calling the **`compile`** method:

```
model = tf.keras.Sequential([
# Adds a densely-connected layer with 64 units to the model:
layers.Dense(64, activation='relu', input_shape=(32,)),
# Add another:
layers.Dense(64, activation='relu'),
# Add a softmax layer with 10 output units:
```

```
layers.Dense(10, activation='softmax')])

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

tf.keras.Model.compile (https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile) takes three important arguments:

- **optimizer**: This object specifies the training procedure. Pass it optimizer instances from the **tf.train** (https://www.tensorflow.org/api_docs/python/tf/train) module, such as **tf.train.AdamOptimizer** (https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer), **tf.train.RMSPropOptimizer** (https://www.tensorflow.org/api_docs/python/tf/train/RMSPropOptimizer), or **tf.train.GradientDescentOptimizer** (https://www.tensorflow.org/api_docs/python/tf/train/GradientDescentOptimizer).
- **loss**: The function to minimize during optimization. Common choices include mean square error (**mse**), **categorical_crossentropy**, and **binary_crossentropy**. Loss functions are specified by name or by passing a callable object from the **tf.keras.losses** (https://www.tensorflow.org/api_docs/python/tf/keras/losses) module.
- **metrics**: Used to monitor training. These are string names or callables from the **tf.keras.metrics** (https://www.tensorflow.org/api_docs/python/tf/keras/metrics) module.

The following shows a few examples of configuring a model for training:

```
# Configure a model for mean-squared error regression.
model.compile(optimizer=tf.train.AdamOptimizer(0.01),
              loss='mse',       # mean squared error
              metrics=['mae'])  # mean absolute error

# Configure a model for categorical classification.
model.compile(optimizer=tf.train.RMSPropOptimizer(0.01),
              loss=tf.keras.losses.categorical_crossentropy,
              metrics=[tf.keras.metrics.categorical_accuracy])
```

Input NumPy data

For small datasets, use in-memory NumPy (https://www.numpy.org/) arrays to train and evaluate a model. The model is "fit" to the training data using the `fit` method:

```
import numpy as np

def random_one_hot_labels(shape):
  n, n_class = shape
  classes = np.random.randint(0, n_class, n)
  labels = np.zeros((n, n_class))
  labels[np.arange(n), classes] = 1
  return labels

data = np.random.random((1000, 32))
labels = random_one_hot_labels((1000, 10))

model.fit(data, labels, epochs=10, batch_size=32)
```

```
W0708 23:41:18.495157 140374570354432 deprecation.py:506] From /tmpfs/src/tf_do
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the

Epoch 1/10
1000/1000 [==============================] - 1s 808us/sample - loss: 2.3293 - c
Epoch 2/10
1000/1000 [==============================] - 0s 75us/sample - loss: 2.3143 - ca
Epoch 3/10
1000/1000 [==============================] - 0s 75us/sample - loss: 2.3071 - ca
Epoch 4/10
1000/1000 [==============================] - 0s 74us/sample - loss: 2.3054 - ca
Epoch 5/10
```

`tf.keras.Model.fit` (https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) takes three important arguments:

- **epochs**: Training is structured into *epochs*. An epoch is one iteration over the entire input data (this is done in smaller batches).

- **`batch_size`**: When passed NumPy data, the model slices the data into smaller batches and iterates over these batches during training. This integer specifies the size of each batch. Be aware that the last batch may be smaller if the total number of samples is not divisible by the batch size.
- **`validation_data`**: When prototyping a model, you want to easily monitor its performance on some validation data. Passing this argument—a tuple of inputs and labels—allows the model to display the loss and metrics in inference mode for the passed data, at the end of each epoch.

Here's an example using **`validation_data`**:

```
import numpy as np

data = np.random.random((1000, 32))
labels = random_one_hot_labels((1000, 10))

val_data = np.random.random((100, 32))
val_labels = random_one_hot_labels((100, 10))

model.fit(data, labels, epochs=10, batch_size=32,
          validation_data=(val_data, val_labels))
```

```
Train on 1000 samples, validate on 100 samples
Epoch 1/10
1000/1000 [==============================] - 0s 140us/sample - loss: 2.3373 - c
Epoch 2/10
1000/1000 [==============================] - 0s 88us/sample - loss: 2.3043 - ca
Epoch 3/10
1000/1000 [==============================] - 0s 88us/sample - loss: 2.2883 - ca
Epoch 4/10
1000/1000 [==============================] - 0s 83us/sample - loss: 2.2822 - ca
Epoch 5/10
1000/1000 [==============================] - 0s 83us/sample - loss: 2.2662 - ca
Epoch 6/10
1000/1000 [==============================] - 0s 80us/sample - loss: 2.2410 - ca
```

Input tf.data datasets

Use the Datasets API (https://www.tensorflow.org/guide/datasets) to scale to large datasets or multi-device training. Pass a **tf.data.Dataset** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) instance to the `fit` method:

```
# Instantiates a toy dataset instance:
dataset = tf.data.Dataset.from_tensor_slices((data, labels))
dataset = dataset.batch(32)
dataset = dataset.repeat()

# Don't forget to specify `steps_per_epoch` when calling `fit` on a dataset.
model.fit(dataset, epochs=10, steps_per_epoch=30)
```

```
W0708 23:41:23.003233 140374570354432 training_utils.py:1300] Expected a shuffl

Epoch 1/10
30/30 [==============================] - 0s 5ms/step - loss: 2.0537 - categoric
Epoch 2/10
30/30 [==============================] - 0s 2ms/step - loss: 2.0128 - categoric
Epoch 3/10
30/30 [==============================] - 0s 3ms/step - loss: 1.9480 - categoric
Epoch 4/10
30/30 [==============================] - 0s 3ms/step - loss: 1.8926 - categoric
Epoch 5/10
30/30 [==============================] - 0s 2ms/step - loss: 1.8420 - categoric
Epoch 6/10
30/30 [==============================] - 0s 2ms/step - loss: 1.7617 - categoric
```

Here, the `fit` method uses the `steps_per_epoch` argument—this is the number of training steps the model runs before it moves to the next epoch. Since the `Dataset` yields batches of data, this snippet does not require a `batch_size`.

Datasets can also be used for validation:

```
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)
```

```
W0708 23:41:23.944267 140374570354432 training_utils.py:1300] Expected a shuffl

Epoch 1/10
30/30 [==============================] - 0s 8ms/step - loss: 1.5234 - categoric
Epoch 2/10
30/30 [==============================] - 0s 3ms/step - loss: 1.4501 - categoric
Epoch 3/10
30/30 [==============================] - 0s 3ms/step - loss: 1.4036 - categoric
Epoch 4/10
30/30 [==============================] - 0s 3ms/step - loss: 1.3496 - categoric
Epoch 5/10
30/30 [==============================] - 0s 3ms/step - loss: 1.3049 - categoric
Epoch 6/10
```

Evaluate and predict

The **`tf.keras.Model.evaluate`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model#evaluate) and **`tf.keras.Model.predict`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model#predict) methods can use NumPy data and a **`tf.data.Dataset`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset).

To *evaluate* the inference-mode loss and metrics for the data provided:

```
data = np.random.random((1000, 32))
labels = random_one_hot_labels((1000, 10))

model.evaluate(data, labels, batch_size=32)

model.evaluate(dataset, steps=30)
```

```
1000/1000 [==============================] - 0s 100us/sample - loss: 4.0635 - ca
30/30 [==============================] - 0s 4ms/step - loss: 1.2802 - categorica

[1.2801913340886435, 0.534375]
```

And to *predict* the output of the last layer in inference for the data provided, as a NumPy array:

```
result = model.predict(data, batch_size=32)
print(result.shape)
```

```
(1000, 10)
```

Build advanced models

Functional API

The **`tf.keras.Sequential`** (https://www.tensorflow.org/api_docs/python/tf/keras/Sequential) model is a simple stack of layers that cannot represent arbitrary models. Use the Keras functional API (https://keras.io/getting-started/functional-api-guide/) to build complex model topologies such as:

- 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`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model) instance.
3. This model is trained just like the **`Sequential`** model.

The following example uses the functional API to build a simple, fully-connected network:

```
inputs = tf.keras.Input(shape=(32,))  # Returns a placeholder tensor

# A layer instance is callable on a tensor, and returns a tensor.
x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)
predictions = layers.Dense(10, activation='softmax')(x)
```

Instantiate the model given inputs and outputs.

```
model = tf.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)
```

```
Epoch 1/5
1000/1000 [==============================] - 0s 147us/sample - loss: 2.3158 - a
Epoch 2/5
1000/1000 [==============================] - 0s 73us/sample - loss: 2.3136 - ac
Epoch 3/5
1000/1000 [==============================] - 0s 74us/sample - loss: 2.3113 - ac
Epoch 4/5
1000/1000 [==============================] - 0s 78us/sample - loss: 2.3043 - ac
Epoch 5/5
1000/1000 [==============================] - 0s 74us/sample - loss: 2.2930 - ac

<tensorflow.python.keras.callbacks.History at 0x7faafa1a6588>
```

Model subclassing

Build a fully-customizable model by subclassing **`tf.keras.Model`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model) and defining your own forward pass. Create layers in the **`__init__`** method and set them as attributes of the class instance. Define the forward pass in the **`call`** method.

Model subclassing is particularly useful when eager execution (https://www.tensorflow.org/guide/eager) is enabled since the forward pass can be written imperatively.

Key Point: Use the right API for the job. While model subclassing offers flexibility, it comes at a cost of greater complexity and more opportunities for user errors. If possible, prefer the functional API.

The following example shows a subclassed **`tf.keras.Model`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model) using a custom forward pass:

```
class MyModel(tf.keras.Model):

  def __init__(self, num_classes=10):
    super(MyModel, self).__init__(name='my_model')
    self.num_classes = num_classes
    # Define your layers here.
    self.dense_1 = layers.Dense(32, activation='relu')
    self.dense_2 = layers.Dense(num_classes, activation='sigmoid')

  def call(self, inputs):
    # Define your forward pass here,
    # using layers you previously defined (in `__init__`).
    x = self.dense_1(inputs)
    return self.dense_2(x)

  def compute_output_shape(self, input_shape):
    # You need to override this function if you want to use the subclassed model
    # as part of a functional-style model.
    # Otherwise, this method is optional.
    shape = tf.TensorShape(input_shape).as_list()
```

```
    shape[-1] = self.num_classes
    return tf.TensorShape(shape)
```

Instantiate the new model class:

```
model = MyModel(num_classes=10)

# 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)
```

```
W0708 23:41:27.032992 140374570354432 deprecation.py:323] From /tmpfs/src/tf_do
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where

Epoch 1/5
1000/1000 [==============================] - 0s 165us/sample - loss: 2.3270 - a
Epoch 2/5
1000/1000 [==============================] - 0s 83us/sample - loss: 2.3238 - ac
Epoch 3/5
1000/1000 [==============================] - 0s 86us/sample - loss: 2.3164 - ac
Epoch 4/5
1000/1000 [==============================] - 0s 91us/sample - loss: 2.3096 - ac
Epoch 5/5
1000/1000 [==============================] - 0s 87us/sample - loss: 2.3044 - ac
```

Custom layers

Create a custom layer by subclassing [`tf.keras.layers.Layer`](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Layer) (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Layer) and implementing 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.

Here's an example of a custom layer that implements a **`matmul`** of an input with a kernel matrix:

```
class MyLayer(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)
    # Make sure to call the `build` method 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
    return base_config

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

Create a model using your custom layer:

```
model = tf.keras.Sequential([
    MyLayer(10),
    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, labels, batch_size=32, epochs=5)
```

```
W0708 23:41:27.883049 140374570354432 deprecation.py:506] From /tmpfs/src/tf_do
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the

Epoch 1/5
1000/1000 [==============================] - 0s 124us/sample - loss: 2.3102 - a
Epoch 2/5
1000/1000 [==============================] - 0s 67us/sample - loss: 2.3085 - ac
Epoch 3/5
1000/1000 [==============================] - 0s 70us/sample - loss: 2.3057 - ac
Epoch 4/5
1000/1000 [==============================] - 0s 65us/sample - loss: 2.3018 - ac
Epoch 5/5
```

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`** (https://www.tensorflow.org/api_docs/python/tf/keras/callbacks) that include:

- **`tf.keras.callbacks.ModelCheckpoint`** (https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint):

Save checkpoints of your model at regular intervals.

- **`tf.keras.callbacks.LearningRateScheduler`** (https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/LearningRateScheduler) : Dynamically change the learning rate.
- **`tf.keras.callbacks.EarlyStopping`** (https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping): Interrupt training when validation performance has stopped improving.
- **`tf.keras.callbacks.TensorBoard`** (https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/TensorBoard): Monitor the model's behavior using TensorBoard (https://www.tensorflow.org/guide/summaries_and_tensorboard).

To use a **`tf.keras.callbacks.Callback`** (https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/Callback), pass it to the model's `fit` method:

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

```
Train on 1000 samples, validate on 100 samples
Epoch 1/5
1000/1000 [==============================] - 0s 155us/sample - loss: 2.2967 - ac
Epoch 2/5
1000/1000 [==============================] - 0s 78us/sample - loss: 2.2947 - acc
Epoch 3/5
1000/1000 [==============================] - 0s 79us/sample - loss: 2.2920 - acc

<tensorflow.python.keras.callbacks.History at 0x7faaf03cbc88>
```

Save and restore

Weights only

Save and load the weights of a model using **tf.keras.Model.save_weights** (https://www.tensorflow.org/api_docs/python/tf/keras/Model#save_weights):

```
model = tf.keras.Sequential([
layers.Dense(64, activation='relu', input_shape=(32,)),
layers.Dense(10, activation='softmax')])

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

```
# Save weights to a TensorFlow Checkpoint file
model.save_weights('./weights/my_model')

# Restore the model's state,
# this requires a model with the same architecture.
model.load_weights('./weights/my_model')
```

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7faae865db70
```

By default, this saves the model's weights in the TensorFlow checkpoint (https://www.tensorflow.org/guide/checkpoints) file format. Weights can also be saved to the Keras HDF5 format (the default for the multi-backend implementation of Keras):

```
# Save weights to a HDF5 file
model.save_weights('my_model.h5', save_format='h5')

# Restore the model's state
model.load_weights('my_model.h5')
```

Configuration only

A model's configuration can be saved—this serializes the model architecture without any weights. A saved configuration can recreate and initialize the same model, even without the code that defined the original model. Keras supports JSON and YAML serialization formats:

```
# Serialize a model to JSON format
json_string = model.to_json()
json_string
```


```
'{"class_name": "Sequential", "config": {"name": "sequential_3", "layers": [{"cl
```


```
import json
import pprint
pprint.pprint(json.loads(json_string))
```


```
{'backend': 'tensorflow',
 'class_name': 'Sequential',
 'config': {'layers': [{'class_name': 'Dense',
                        'config': {'activation': 'relu',
                                   'activity_regularizer': None,
                                   'batch_input_shape': [None, 32],
                                   'bias_constraint': None,
                                   'bias_initializer': {'class_name': 'Zeros',
                                                        'config': {'dtype': 'fl
                                   'bias_regularizer': None,
                                   'dtype': 'float32',
                                   'kernel_constraint': None,
                                   'kernel_initializer': {'class_name': 'Glorot
```


Recreate the model (newly initialized) from the JSON:

```
fresh_model = tf.keras.models.model_from_json(json_string)
```

```
W0708 23:41:29.795922 140374570354432 deprecation.py:506] From /tmpfs/src/tf_doc
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the c
W0708 23:41:29.797740 140374570354432 deprecation.py:506] From /tmpfs/src/tf_doc
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the c
```

Serializing a model to YAML format requires that you install `pyyaml` *before you import TensorFlow*:

```
yaml_string = model.to_yaml()
print(yaml_string)
```


```
backend: tensorflow
class_name: Sequential
config:
  layers:
  - class_name: Dense
    config:
      activation: relu
      activity_regularizer: null
      batch_input_shape: !!python/tuple [null, 32]
      bias_constraint: null
      bias_initializer:
        class_name: Zeros
        config: {dtype: float32}
      bias_regularizer: null
```


Recreate the model from the YAML:

```
fresh_model = tf.keras.models.model_from_yaml(yaml_string)
```

Caution: Subclassed models are not serializable because their architecture is defined by the Python code in the body of the `call` method.

Entire model

The entire model can be saved to a file that contains the weight values, the model's configuration, and even the optimizer's configuration. This allows you to checkpoint a model and resume training later—from the exact same state—without access to the original code.

```
# Create a trivial model
model = tf.keras.Sequential([
  layers.Dense(64, activation='relu', input_shape=(32,)),
  layers.Dense(10, activation='softmax')
])
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels, batch_size=32, epochs=5)


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

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

```
Epoch 1/5
1000/1000 [==============================] - 0s 164us/sample - loss: 2.3368 - ac
Epoch 2/5
1000/1000 [==============================] - 0s 81us/sample - loss: 2.3098 - acc
Epoch 3/5
1000/1000 [==============================] - 0s 76us/sample - loss: 2.3001 - acc
Epoch 4/5
1000/1000 [==============================] - 0s 77us/sample - loss: 2.2924 - acc
Epoch 5/5
1000/1000 [==============================] - 0s 80us/sample - loss: 2.2876 - acc
```

Eager execution

Eager execution (https://www.tensorflow.org/guide/eager) is an imperative programming environment that evaluates operations immediately. This is not required for Keras, but is supported by **`tf.keras`** (https://www.tensorflow.org/api_docs/python/tf/keras) and useful for inspecting your program and debugging.

All of the **`tf.keras`** (https://www.tensorflow.org/api_docs/python/tf/keras) model-building APIs are compatible with eager execution. And while the `Sequential` and functional APIs can be used, eager execution especially benefits *model subclassing* and building *custom layers*—the APIs that require you to write the forward pass as code (instead of the APIs that create models by assembling existing layers).

See the eager execution guide (https://www.tensorflow.org/guide/eager#build_a_model) for examples of using Keras models with custom training loops and **`tf.GradientTape`** (https://www.tensorflow.org/api_docs/python/tf/GradientTape).

Distribution

Estimators

The Estimators (https://www.tensorflow.org/guide/estimators) API is used for training models for distributed environments. This targets industry use cases such as distributed training on large datasets that can export a model for production.

A **`tf.keras.Model`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model) can be trained with the **`tf.estimator`** (https://www.tensorflow.org/api_docs/python/tf/estimator) API by converting the model to an **`tf.estimator.Estimator`** (https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator) object with **`tf.keras.estimator.model_to_estimator`** (https://www.tensorflow.org/api_docs/python/tf/keras/estimator/model_to_estimator). See Creating Estimators from Keras models (https://www.tensorflow.org/guide/estimators#creating_estimators_from_keras_models).

```
model = tf.keras.Sequential([layers.Dense(64, activation='relu', input_shape=(32
                          layers.Dense(10,activation='softmax')])

model.compile(optimizer=tf.train.RMSPropOptimizer(0.001),
```

```
              loss='categorical_crossentropy',
              metrics=['accuracy'])

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

```
W0708 23:41:32.058223 140374570354432 estimator.py:1811] Using temporary folder
```

Note: Enable eager execution (https://www.tensorflow.org/guide/eager) for debugging Estimator input functions (https://www.tensorflow.org/guide/premade_estimators#create_input_functions) and inspecting data.

Multiple GPUs

`tf.keras` (https://www.tensorflow.org/api_docs/python/tf/keras) models can run on multiple GPUs using **`tf.contrib.distribute.DistributionStrategy`** (https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy). This API provides distributed training on multiple GPUs with almost no changes to existing code.

Currently, **`tf.contrib.distribute.MirroredStrategy`** (https://www.tensorflow.org/api_docs/python/tf/contrib/distribute/MirroredStrategy) is the only supported distribution strategy. `MirroredStrategy` does in-graph replication with synchronous training using all-reduce on a single machine. To use `DistributionStrategy` with Keras, convert the **`tf.keras.Model`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model) to a **`tf.estimator.Estimator`** (https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator) with **`tf.keras.estimator.model_to_estimator`** (https://www.tensorflow.org/api_docs/python/tf/keras/estimator/model_to_estimator), then train the estimator

The following example distributes a **`tf.keras.Model`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model) across multiple GPUs on a single machine.

First, define a simple model:

```
model = tf.keras.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10,)))
model.add(layers.Dense(1, activation='sigmoid'))

optimizer = tf.train.GradientDescentOptimizer(0.2)

model.compile(loss='binary_crossentropy', optimizer=optimizer)
model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_23 (Dense)             (None, 16)                176
_________________________________________________________________
dense_24 (Dense)             (None, 1)                 17
=================================================================
Total params: 193
Trainable params: 193
Non-trainable params: 0
_________________________________________________________________
```

Define an *input pipeline*. The **`input_fn`** returns a **`tf.data.Dataset`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) object used to distribute the data across multiple devices—with each device processing a slice of the input batch.

```
def input_fn():
  x = np.random.random((1024, 10))
  y = np.random.randint(2, size=(1024, 1))
  x = tf.cast(x, tf.float32)
  dataset = tf.data.Dataset.from_tensor_slices((x, y))
  dataset = dataset.repeat(10)
  dataset = dataset.batch(32)
  return dataset
```

Next, create a **tf.estimator.RunConfig** (https://www.tensorflow.org/api_docs/python/tf/estimator/RunConfig) and set the **train_distribute** argument to the **tf.contrib.distribute.MirroredStrategy** (https://www.tensorflow.org/api_docs/python/tf/contrib/distribute/MirroredStrategy) instance. When creating **MirroredStrategy**, you can specify a list of devices or set the **num_gpus** argument. The default uses all available GPUs, like the following:

```
strategy = tf.contrib.distribute.MirroredStrategy()
config = tf.estimator.RunConfig(train_distribute=strategy)
```

```
W0708 23:41:33.447634 140374570354432 lazy_loader.py:50]
The TensorFlow contrib module will not be included in TensorFlow 2.0.
For more information, please see:
  * https://github.com/tensorflow/community/blob/master/rfcs/20180907-contrib-su
  * https://github.com/tensorflow/addons
  * https://github.com/tensorflow/io (for I/O related ops)
If you depend on functionality not listed there, please file an issue.
```

Convert the Keras model to a **tf.estimator.Estimator** (https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator) instance:

```
keras_estimator = tf.keras.estimator.model_to_estimator(
  keras_model=model,
  config=config,
  model_dir='/tmp/model_dir')
```

Finally, train the **Estimator** instance by providing the **input_fn** and **steps** arguments:

```
keras_estimator.train(input_fn=input_fn, steps=10)
```

```
<tensorflow_estimator.python.estimator.estimator.Estimator at 0x7fa5c530e5f8>
```