


Keras

 Star

55,962

[About Keras](#)

[Getting started](#)

Developer guides

The Functional API

The Sequential model

Making new layers & models via subclassing

Training & evaluation with the built-in methods

Customizing what happens in `fit()`

Writing a training loop from scratch

Serialization & saving

Writing your own callbacks

Working with preprocessing layers

Working with recurrent neural networks

Understanding masking & padding

Multi-GPU & distributed training

Transfer learning & fine-tuning

Hyperparameter Tuning

KerasCV

KerasNLP

[Keras API reference](#)

[Code examples](#)

[Why choose Keras?](#)

[Community & governance](#)

[Contributing to Keras](#)

[KerasTuner](#)

[KerasCV](#)

[KerasNLP](#)

» [Developer guides](#) / Writing your own callbacks

Writing your own callbacks

Authors: Rick Chao, Francois Chollet
Date created: 2019/03/20
Last modified: 2020/07/12
Description: Complete guide to writing new Keras callbacks.

 [View in Colab](#) •  [GitHub source](#)

Introduction

A callback is a powerful tool to customize the behavior of a Keras model during training, evaluation, or inference. Examples include [tf.keras.callbacks.TensorBoard](#) to visualize training progress and results with TensorBoard, or [tf.keras.callbacks.ModelCheckpoint](#) to periodically save your model during training.

In this guide, you will learn what a Keras callback is, what it can do, and how you can build your own. We provide a few demos of simple callback applications to get you started.

Setup

```
import tensorflow as tf
from tensorflow import keras
```

Keras callbacks overview

All callbacks subclass the [keras.callbacks.Callback](#) class, and override a set of methods called at various stages of training, testing, and predicting. Callbacks are useful to get a view on internal states and statistics of the model during training.

You can pass a list of callbacks (as the keyword argument `callbacks`) to the following model methods:

- `keras.Model.fit()`
- `keras.Model.evaluate()`
- `keras.Model.predict()`

An overview of callback methods

Global methods

`on_(train|test|predict)_begin(self, logs=None)`

Called at the beginning of `fit/evaluate/predict`.

`on_(train|test|predict)_end(self, logs=None)`

Called at the end of `fit/evaluate/predict`.

Batch-level methods for training/testing/predicting

`on_(train|test|predict)_batch_begin(self, batch, logs=None)`

Called right before processing a batch during training/testing/predicting.

`on_(train|test|predict)_batch_end(self, batch, logs=None)`

Called at the end of training/testing/predicting a batch. Within this method, `logs` is a dict containing the metrics results.

Epoch-level methods (training only)

`on_epoch_begin(self, epoch, logs=None)`

Called at the beginning of an epoch during training.

`on_epoch_end(self, epoch, logs=None)`

Called at the end of an epoch during training.

A basic example

Let's take a look at a concrete example. To get started, let's import tensorflow and define a simple Sequential Keras model:

```
# Define the Keras model to add callbacks to
def get_model():
    model = keras.Sequential()
    model.add(keras.layers.Dense(1, input_dim=784))
    model.compile(
        optimizer=keras.optimizers.RMSprop(learning_rate=0.1),
        loss="mean_squared_error",
        metrics=["mean_absolute_error"],
    )
    return model
```

Then, load the MNIST data for training and testing from Keras datasets API:

```
# Load example MNIST data and pre-process it
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape(-1, 784).astype("float32") / 255.0
x_test = x_test.reshape(-1, 784).astype("float32") / 255.0

# Limit the data to 1000 samples
x_train = x_train[:1000]
y_train = y_train[:1000]
x_test = x_test[:1000]
y_test = y_test[:1000]
```

Now, define a simple custom callback that logs:

- When `fit/evaluate/predict` starts & ends
- When each epoch starts & ends
- When each training batch starts & ends
- When each evaluation (test) batch starts & ends
- When each inference (prediction) batch starts & ends

```

class CustomCallback(keras.callbacks.Callback):
    def on_train_begin(self, logs=None):
        keys = list(logs.keys())
        print("Starting training; got log keys: {}".format(keys))

    def on_train_end(self, logs=None):
        keys = list(logs.keys())
        print("Stop training; got log keys: {}".format(keys))

    def on_epoch_begin(self, epoch, logs=None):
        keys = list(logs.keys())
        print("Start epoch {} of training; got log keys: {}".format(epoch, keys))

    def on_epoch_end(self, epoch, logs=None):
        keys = list(logs.keys())
        print("End epoch {} of training; got log keys: {}".format(epoch, keys))

    def on_test_begin(self, logs=None):
        keys = list(logs.keys())
        print("Start testing; got log keys: {}".format(keys))

    def on_test_end(self, logs=None):
        keys = list(logs.keys())
        print("Stop testing; got log keys: {}".format(keys))

    def on_predict_begin(self, logs=None):
        keys = list(logs.keys())
        print("Start predicting; got log keys: {}".format(keys))

    def on_predict_end(self, logs=None):
        keys = list(logs.keys())
        print("Stop predicting; got log keys: {}".format(keys))

    def on_train_batch_begin(self, batch, logs=None):
        keys = list(logs.keys())
        print("...Training: start of batch {}; got log keys: {}".format(batch, keys))

    def on_train_batch_end(self, batch, logs=None):
        keys = list(logs.keys())
        print("...Training: end of batch {}; got log keys: {}".format(batch, keys))

    def on_test_batch_begin(self, batch, logs=None):
        keys = list(logs.keys())
        print("...Evaluating: start of batch {}; got log keys: {}".format(batch, keys))

    def on_test_batch_end(self, batch, logs=None):
        keys = list(logs.keys())
        print("...Evaluating: end of batch {}; got log keys: {}".format(batch, keys))

    def on_predict_batch_begin(self, batch, logs=None):
        keys = list(logs.keys())
        print("...Predicting: start of batch {}; got log keys: {}".format(batch, keys))

    def on_predict_batch_end(self, batch, logs=None):
        keys = list(logs.keys())
        print("...Predicting: end of batch {}; got log keys: {}".format(batch, keys))

```

Let's try it out:

```

model = get_model()
model.fit(
    x_train,
    y_train,
    batch_size=128,
    epochs=1,
    verbose=0,
    validation_split=0.5,
    callbacks=[CustomCallback()],
)

res = model.evaluate(
    x_test, y_test, batch_size=128, verbose=0, callbacks=[CustomCallback()]
)

res = model.predict(x_test, batch_size=128, callbacks=[CustomCallback()])

```

```

Starting training; got log keys: []
Start epoch 0 of training; got log keys: []
...Training: start of batch 0; got log keys: []
...Training: end of batch 0; got log keys: ['loss', 'mean_absolute_error']
...Training: start of batch 1; got log keys: []
...Training: end of batch 1; got log keys: ['loss', 'mean_absolute_error']
...Training: start of batch 2; got log keys: []
...Training: end of batch 2; got log keys: ['loss', 'mean_absolute_error']
...Training: start of batch 3; got log keys: []
...Training: end of batch 3; got log keys: ['loss', 'mean_absolute_error']
Start testing; got log keys: []
...Evaluating: start of batch 0; got log keys: []
...Evaluating: end of batch 0; got log keys: ['loss', 'mean_absolute_error']
...Evaluating: start of batch 1; got log keys: []
...Evaluating: end of batch 1; got log keys: ['loss', 'mean_absolute_error']
...Evaluating: start of batch 2; got log keys: []
...Evaluating: end of batch 2; got log keys: ['loss', 'mean_absolute_error']
...Evaluating: start of batch 3; got log keys: []
...Evaluating: end of batch 3; got log keys: ['loss', 'mean_absolute_error']
Stop testing; got log keys: ['loss', 'mean_absolute_error']
End epoch 0 of training; got log keys: ['loss', 'mean_absolute_error', 'val_loss',
'val_mean_absolute_error']
Stop training; got log keys: ['loss', 'mean_absolute_error', 'val_loss',
'val_mean_absolute_error']
Start testing; got log keys: []
...Evaluating: start of batch 0; got log keys: []
...Evaluating: end of batch 0; got log keys: ['loss', 'mean_absolute_error']
...Evaluating: start of batch 1; got log keys: []
...Evaluating: end of batch 1; got log keys: ['loss', 'mean_absolute_error']
...Evaluating: start of batch 2; got log keys: []
...Evaluating: end of batch 2; got log keys: ['loss', 'mean_absolute_error']
...Evaluating: start of batch 3; got log keys: []
...Evaluating: end of batch 3; got log keys: ['loss', 'mean_absolute_error']
...Evaluating: start of batch 4; got log keys: []
...Evaluating: end of batch 4; got log keys: ['loss', 'mean_absolute_error']
...Evaluating: start of batch 5; got log keys: []
...Evaluating: end of batch 5; got log keys: ['loss', 'mean_absolute_error']
...Evaluating: start of batch 6; got log keys: []
...Evaluating: end of batch 6; got log keys: ['loss', 'mean_absolute_error']
...Evaluating: start of batch 7; got log keys: []
...Evaluating: end of batch 7; got log keys: ['loss', 'mean_absolute_error']
Stop testing; got log keys: ['loss', 'mean_absolute_error']
Start predicting; got log keys: []
...Predicting: start of batch 0; got log keys: []
...Predicting: end of batch 0; got log keys: ['outputs']
...Predicting: start of batch 1; got log keys: []
...Predicting: end of batch 1; got log keys: ['outputs']
...Predicting: start of batch 2; got log keys: []
...Predicting: end of batch 2; got log keys: ['outputs']
...Predicting: start of batch 3; got log keys: []
...Predicting: end of batch 3; got log keys: ['outputs']
...Predicting: start of batch 4; got log keys: []
...Predicting: end of batch 4; got log keys: ['outputs']
...Predicting: start of batch 5; got log keys: []
...Predicting: end of batch 5; got log keys: ['outputs']
...Predicting: start of batch 6; got log keys: []
...Predicting: end of batch 6; got log keys: ['outputs']
...Predicting: start of batch 7; got log keys: []
...Predicting: end of batch 7; got log keys: ['outputs']
Stop predicting; got log keys: []

```

Usage of **logs** dict

The **logs** dict contains the loss value, and all the metrics at the end of a batch or epoch. Example includes the loss and mean absolute error.

```
class LossAndErrorPrintingCallback(keras.callbacks.Callback):
    def on_train_batch_end(self, batch, logs=None):
        print(
            "Up to batch {}, the average loss is {:.2f}.".format(batch, logs["loss"])
        )

    def on_test_batch_end(self, batch, logs=None):
        print(
            "Up to batch {}, the average loss is {:.2f}.".format(batch, logs["loss"])
        )

    def on_epoch_end(self, epoch, logs=None):
        print(
            "The average loss for epoch {} is {:.2f} "
            "and mean absolute error is {:.2f}.".format(
                epoch, logs["loss"], logs["mean_absolute_error"]
            )
        )

model = get_model()
model.fit(
    x_train,
    y_train,
    batch_size=128,
    epochs=2,
    verbose=0,
    callbacks=[LossAndErrorPrintingCallback()],
)

res = model.evaluate(
    x_test,
    y_test,
    batch_size=128,
    verbose=0,
    callbacks=[LossAndErrorPrintingCallback()],
)
```

```
Up to batch 0, the average loss is   30.30.
Up to batch 1, the average loss is  462.85.
Up to batch 2, the average loss is  316.49.
Up to batch 3, the average loss is  239.75.
Up to batch 4, the average loss is  193.43.
Up to batch 5, the average loss is  162.31.
Up to batch 6, the average loss is  139.97.
Up to batch 7, the average loss is  125.99.
The average loss for epoch 0 is  125.99 and mean absolute error is    6.00.
Up to batch 0, the average loss is    4.34.
Up to batch 1, the average loss is    4.29.
Up to batch 2, the average loss is    4.17.
Up to batch 3, the average loss is    4.39.
Up to batch 4, the average loss is    4.33.
Up to batch 5, the average loss is    4.24.
Up to batch 6, the average loss is    4.34.
Up to batch 7, the average loss is    4.40.
The average loss for epoch 1 is    4.40 and mean absolute error is    1.68.
Up to batch 0, the average loss is    5.63.
Up to batch 1, the average loss is    5.24.
Up to batch 2, the average loss is    5.18.
Up to batch 3, the average loss is    5.12.
Up to batch 4, the average loss is    5.25.
Up to batch 5, the average loss is    5.30.
Up to batch 6, the average loss is    5.23.
Up to batch 7, the average loss is    5.17.
```

Usage of `self.model` attribute

In addition to receiving log information when one of their methods is called, callbacks have access to the model associated with the current round of training/evaluation/inference: `self.model`.

Here are a few of the things you can do with `self.model` in a callback:

- Set `self.model.stop_training = True` to immediately interrupt training.
- Mutate hyperparameters of the optimizer (available as `self.model.optimizer`), such as `self.model.optimizer.learning_rate`.
- Save the model at period intervals.
- Record the output of `model.predict()` on a few test samples at the end of each epoch, to use as a sanity check during training.
- Extract visualizations of intermediate features at the end of each epoch, to monitor what the model is learning over time.
- etc.

Let's see this in action in a couple of examples.

Examples of Keras callback applications

Early stopping at minimum loss

This first example shows the creation of a `Callback` that stops training when the minimum of loss has been reached, by setting the attribute `self.model.stop_training` (boolean). Optionally, you can provide an argument `patience` to specify how many epochs we should wait before stopping after having reached a local minimum.

[`tf.keras.callbacks.EarlyStopping`](#) provides a more complete and general implementation.

```

import numpy as np

class EarlyStoppingAtMinLoss(keras.callbacks.Callback):
    """Stop training when the loss is at its min, i.e. the loss stops decreasing.

    Arguments:
        patience: Number of epochs to wait after min has been hit. After this
        number of no improvement, training stops.
    """

    def __init__(self, patience=0):
        super(EarlyStoppingAtMinLoss, self).__init__()
        self.patience = patience
        # best_weights to store the weights at which the minimum loss occurs.
        self.best_weights = None

    def on_train_begin(self, logs=None):
        # The number of epoch it has waited when loss is no longer minimum.
        self.wait = 0
        # The epoch the training stops at.
        self.stopped_epoch = 0
        # Initialize the best as infinity.
        self.best = np.Inf

    def on_epoch_end(self, epoch, logs=None):
        current = logs.get("loss")
        if np.less(current, self.best):
            self.best = current
            self.wait = 0
            # Record the best weights if current results is better (less).
            self.best_weights = self.model.get_weights()
        else:
            self.wait += 1
            if self.wait >= self.patience:
                self.stopped_epoch = epoch
                self.model.stop_training = True
                print("Restoring model weights from the end of the best epoch.")
                self.model.set_weights(self.best_weights)

    def on_train_end(self, logs=None):
        if self.stopped_epoch > 0:
            print("Epoch %05d: early stopping" % (self.stopped_epoch + 1))

model = get_model()
model.fit(
    x_train,
    y_train,
    batch_size=64,
    steps_per_epoch=5,
    epochs=30,
    verbose=0,
    callbacks=[LossAndErrorPrintingCallback(), EarlyStoppingAtMinLoss()],
)

```



```
Up to batch 0, the average loss is 26.58.
Up to batch 1, the average loss is 354.71.
Up to batch 2, the average loss is 243.70.
Up to batch 3, the average loss is 185.92.
Up to batch 4, the average loss is 149.93.
The average loss for epoch 0 is 149.93 and mean absolute error is 7.53.
Up to batch 0, the average loss is 7.17.
Up to batch 1, the average loss is 6.42.
Up to batch 2, the average loss is 5.93.
Up to batch 3, the average loss is 5.51.
Up to batch 4, the average loss is 5.44.
The average loss for epoch 1 is 5.44 and mean absolute error is 1.82.
Up to batch 0, the average loss is 4.07.
Up to batch 1, the average loss is 4.77.
Up to batch 2, the average loss is 4.38.
Up to batch 3, the average loss is 4.81.
Up to batch 4, the average loss is 4.57.
The average loss for epoch 2 is 4.57 and mean absolute error is 1.73.
Up to batch 0, the average loss is 3.76.
Up to batch 1, the average loss is 4.10.
Up to batch 2, the average loss is 5.31.
Up to batch 3, the average loss is 7.89.
Up to batch 4, the average loss is 14.11.
The average loss for epoch 3 is 14.11 and mean absolute error is 2.96.
Restoring model weights from the end of the best epoch.
Epoch 00004: early stopping

<tensorflow.python.keras.callbacks.History at 0x7f43690baf60>
```

Learning rate scheduling

In this example, we show how a custom Callback can be used to dynamically change the learning rate of the optimizer during the course of training.

See `callbacks.LearningRateScheduler` for a more general implementations.


```

class CustomLearningRateScheduler(keras.callbacks.Callback):
    """Learning rate scheduler which sets the learning rate according to schedule.

    Arguments:
        schedule: a function that takes an epoch index
                  (integer, indexed from 0) and current learning rate
                  as inputs and returns a new learning rate as output (float).
    """

    def __init__(self, schedule):
        super(CustomLearningRateScheduler, self).__init__()
        self.schedule = schedule

    def on_epoch_begin(self, epoch, logs=None):
        if not hasattr(self.model.optimizer, "lr"):
            raise ValueError('Optimizer must have a "lr" attribute.')
        # Get the current learning rate from model's optimizer.
        lr = float(tf.keras.backend.get_value(self.model.optimizer.learning_rate))
        # Call schedule function to get the scheduled learning rate.
        scheduled_lr = self.schedule(epoch, lr)
        # Set the value back to the optimizer before this epoch starts
        tf.keras.backend.set_value(self.model.optimizer.lr, scheduled_lr)
        print("\nEpoch %05d: Learning rate is %6.4f." % (epoch, scheduled_lr))

LR_SCHEDULE = [
    # (epoch to start, learning rate) tuples
    (3, 0.05),
    (6, 0.01),
    (9, 0.005),
    (12, 0.001),
]

def lr_schedule(epoch, lr):
    """Helper function to retrieve the scheduled learning rate based on epoch."""
    if epoch < LR_SCHEDULE[0][0] or epoch > LR_SCHEDULE[-1][0]:
        return lr
    for i in range(len(LR_SCHEDULE)):
        if epoch == LR_SCHEDULE[i][0]:
            return LR_SCHEDULE[i][1]
    return lr

model = get_model()
model.fit(
    x_train,
    y_train,
    batch_size=64,
    steps_per_epoch=5,
    epochs=15,
    verbose=0,
    callbacks=[
        LossAndErrorPrintingCallback(),
        CustomLearningRateScheduler(lr_schedule),
    ],
)

```

```

Epoch 00000: Learning rate is 0.1000.
Up to batch 0, the average loss is 28.80.
Up to batch 1, the average loss is 473.07.
Up to batch 2, the average loss is 326.01.
Up to batch 3, the average loss is 247.08.
Up to batch 4, the average loss is 198.91.
The average loss for epoch 0 is 198.91 and mean absolute error is 8.52.

```

```

Epoch 00001: Learning rate is 0.1000.
Up to batch 0, the average loss is 3.98.
Up to batch 1, the average loss is 6.35.
Up to batch 2, the average loss is 5.81.
Up to batch 3, the average loss is 6.04.
Up to batch 4, the average loss is 6.09.
The average loss for epoch 1 is 6.09 and mean absolute error is 2.04.

```

Epoch 00002: Learning rate is 0.1000.
Up to batch 0, the average loss is 5.20.
Up to batch 1, the average loss is 5.40.
Up to batch 2, the average loss is 5.15.
Up to batch 3, the average loss is 5.02.
Up to batch 4, the average loss is 4.90.
The average loss for epoch 2 is 4.90 and mean absolute error is 1.78.

Epoch 00003: Learning rate is 0.0500.
Up to batch 0, the average loss is 4.70.
Up to batch 1, the average loss is 4.16.
Up to batch 2, the average loss is 4.34.
Up to batch 3, the average loss is 4.34.
Up to batch 4, the average loss is 4.23.
The average loss for epoch 3 is 4.23 and mean absolute error is 1.63.

Epoch 00004: Learning rate is 0.0500.
Up to batch 0, the average loss is 3.02.
Up to batch 1, the average loss is 3.41.
Up to batch 2, the average loss is 3.81.
Up to batch 3, the average loss is 4.13.
Up to batch 4, the average loss is 3.86.
The average loss for epoch 4 is 3.86 and mean absolute error is 1.54.

Epoch 00005: Learning rate is 0.0500.
Up to batch 0, the average loss is 3.75.
Up to batch 1, the average loss is 3.72.
Up to batch 2, the average loss is 3.55.
Up to batch 3, the average loss is 3.78.
Up to batch 4, the average loss is 3.63.
The average loss for epoch 5 is 3.63 and mean absolute error is 1.52.

Epoch 00006: Learning rate is 0.0100.
Up to batch 0, the average loss is 4.61.
Up to batch 1, the average loss is 3.78.
Up to batch 2, the average loss is 4.25.
Up to batch 3, the average loss is 4.15.
Up to batch 4, the average loss is 3.83.
The average loss for epoch 6 is 3.83 and mean absolute error is 1.54.

Epoch 00007: Learning rate is 0.0100.
Up to batch 0, the average loss is 3.91.
Up to batch 1, the average loss is 3.57.
Up to batch 2, the average loss is 3.19.
Up to batch 3, the average loss is 2.92.
Up to batch 4, the average loss is 3.03.
The average loss for epoch 7 is 3.03 and mean absolute error is 1.37.

Epoch 00008: Learning rate is 0.0100.
Up to batch 0, the average loss is 4.11.
Up to batch 1, the average loss is 3.89.
Up to batch 2, the average loss is 3.36.
Up to batch 3, the average loss is 3.12.
Up to batch 4, the average loss is 3.37.
The average loss for epoch 8 is 3.37 and mean absolute error is 1.43.

Epoch 00009: Learning rate is 0.0050.
Up to batch 0, the average loss is 3.31.
Up to batch 1, the average loss is 3.65.
Up to batch 2, the average loss is 3.73.
Up to batch 3, the average loss is 3.52.
Up to batch 4, the average loss is 3.59.
The average loss for epoch 9 is 3.59 and mean absolute error is 1.46.

```
Epoch 00010: Learning rate is 0.0050.
Up to batch 0, the average loss is      3.13.
Up to batch 1, the average loss is      2.96.
Up to batch 2, the average loss is      2.99.
Up to batch 3, the average loss is      3.19.
Up to batch 4, the average loss is      3.16.
The average loss for epoch 10 is      3.16 and mean absolute error is      1.39.
```

```
Epoch 00011: Learning rate is 0.0050.
Up to batch 0, the average loss is      3.04.
Up to batch 1, the average loss is      3.14.
Up to batch 2, the average loss is      3.23.
Up to batch 3, the average loss is      3.32.
Up to batch 4, the average loss is      3.44.
The average loss for epoch 11 is      3.44 and mean absolute error is      1.43.
```

```
Epoch 00012: Learning rate is 0.0010.
Up to batch 0, the average loss is      2.82.
Up to batch 1, the average loss is      3.41.
Up to batch 2, the average loss is      3.28.
Up to batch 3, the average loss is      3.34.
Up to batch 4, the average loss is      3.28.
The average loss for epoch 12 is      3.28 and mean absolute error is      1.38.
```

```
Epoch 00013: Learning rate is 0.0010.
Up to batch 0, the average loss is      2.67.
Up to batch 1, the average loss is      3.34.
Up to batch 2, the average loss is      3.60.
Up to batch 3, the average loss is      3.49.
Up to batch 4, the average loss is      3.57.
The average loss for epoch 13 is      3.57 and mean absolute error is      1.44.
```

```
Epoch 00014: Learning rate is 0.0010.
Up to batch 0, the average loss is      2.54.
Up to batch 1, the average loss is      2.45.
Up to batch 2, the average loss is      2.65.
Up to batch 3, the average loss is      2.78.
Up to batch 4, the average loss is      2.76.
The average loss for epoch 14 is      2.76 and mean absolute error is      1.33.

<tensorflow.python.keras.callbacks.History at 0x7f4368e52ba8>
```

Built-in Keras callbacks

Be sure to check out the existing Keras callbacks by reading the [API docs](#). Applications include logging to CSV, saving the model, visualizing metrics in TensorBoard, and a lot more!