# **Keras**

<https://keras.io/about/>

<https://keras.io/guides/>

Models API

There are three ways to create Keras models:

The Sequential model, which is very straightforward (a simple list of layers), but is limited to single-input, single-output stacks of layers (as the name gives away).

The Functional API, which is an easy-to-use, fully-featured API that supports arbitrary model architectures. For most people and most use cases, this is what you should be using. This is the Keras "industry strength" model.

Model subclassing, where you implement everything from scratch on your own. Use this if you have complex, out-of-the-box research use cases.

# **The Model class**

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/models/model.py#L32)

### Model class

keras.Model()

A model grouping layers into an object with training/inference features.

There are three ways to instantiate a Model:

## With the "Functional API"

You start from Input, you chain layer calls to specify the model's forward pass, and finally you create your model from inputs and outputs:

inputs = keras.Input(shape=(37,))

x = keras.layers.Dense(32, activation="relu")(inputs)

outputs = keras.layers.Dense(5, activation="softmax")(x)

model = keras.Model(inputs=inputs, outputs=outputs)

Note: Only dicts, lists, and tuples of input tensors are supported. Nested inputs are not supported (e.g. lists of list or dicts of dict).

A new Functional API model can also be created by using the intermediate tensors. This enables you to quickly extract sub-components of the model.

**Example**

inputs = keras.Input(shape=(None, None, 3))

processed = keras.layers.RandomCrop(width=128, height=128)(inputs)

conv = keras.layers.Conv2D(filters=32, kernel\_size=3)(processed)

pooling = keras.layers.GlobalAveragePooling2D()(conv)

feature = keras.layers.Dense(10)(pooling)

full\_model = keras.Model(inputs, feature)

backbone = keras.Model(processed, conv)

activations = keras.Model(conv, feature)

Note that the backbone and activations models are not created with [keras.Input](https://keras.io/api/layers/core_layers/input" \l "input-function) objects, but with the tensors that originate from [keras.Input](https://keras.io/api/layers/core_layers/input" \l "input-function) objects. Under the hood, the layers and weights will be shared across these models, so that user can train the full\_model, and use backbone or activations to do feature extraction. The inputs and outputs of the model can be nested structures of tensors as well, and the created models are standard Functional API models that support all the existing APIs.

## By subclassing the Model class

In that case, you should define your layers in \_\_init\_\_() and you should implement the model's forward pass in call().

class MyModel(keras.Model):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.dense1 = keras.layers.Dense(32, activation="relu")

self.dense2 = keras.layers.Dense(5, activation="softmax")

def call(self, inputs):

x = self.dense1(inputs)

return self.dense2(x)

model = MyModel()

If you subclass Model, you can optionally have a training argument (boolean) in call(), which you can use to specify a different behavior in training and inference:

class MyModel(keras.Model):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.dense1 = keras.layers.Dense(32, activation="relu")

self.dense2 = keras.layers.Dense(5, activation="softmax")

self.dropout = keras.layers.Dropout(0.5)

def call(self, inputs, training=False):

x = self.dense1(inputs)

x = self.dropout(x, training=training)

return self.dense2(x)

model = MyModel()

Once the model is created, you can config the model with losses and metrics with model.compile(), train the model with model.fit(), or use the model to do prediction with model.predict().

## With the Sequential class

In addition, [keras.Sequential](https://keras.io/api/models/sequential" \l "sequential-class) is a special case of model where the model is purely a stack of single-input, single-output layers.

model = keras.Sequential([

keras.Input(shape=(None, None, 3)),

keras.layers.Conv2D(filters=32, kernel\_size=3),

])

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/models/model.py#L216)

### summary method

Model.summary(

line\_length=None,

positions=None,

print\_fn=None,

expand\_nested=False,

show\_trainable=False,

layer\_range=None,

)

Prints a string summary of the network.

**Arguments**

* **line\_length**: Total length of printed lines (e.g. set this to adapt the display to different terminal window sizes).
* **positions**: Relative or absolute positions of log elements in each line. If not provided, becomes [0.3, 0.6, 0.70, 1.]. Defaults to None.
* **print\_fn**: Print function to use. By default, prints to stdout. If stdout doesn't work in your environment, change to print. It will be called on each line of the summary. You can set it to a custom function in order to capture the string summary.
* **expand\_nested**: Whether to expand the nested models. Defaults to False.
* **show\_trainable**: Whether to show if a layer is trainable. Defaults to False.
* **layer\_range**: a list or tuple of 2 strings, which is the starting layer name and ending layer name (both inclusive) indicating the range of layers to be printed in summary. It also accepts regex patterns instead of exact name. In such case, start predicate will be the first element it matches to layer\_range[0] and the end predicate will be the last element it matches to layer\_range[1]. By default None which considers all layers of model.

**Raises**

* **ValueError**: if summary() is called before the model is built.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/models/model.py#L175)

### get\_layer method

Model.get\_layer(name=None, index=None)

Retrieves a layer based on either its name (unique) or index.

If name and index are both provided, index will take precedence. Indices are based on order of horizontal graph traversal (bottom-up).

**Arguments**

* **name**: String, name of layer.
* **index**: Integer, index of layer.

**Returns**

A layer instance.

# **The Sequential class**

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/models/sequential.py#L17)

### Sequential class

keras.Sequential(layers=None, trainable=True, name=None)

Sequential groups a linear stack of layers into a Model.

**Examples**

model = keras.Sequential()

model.add(keras.Input(shape=(16,)))

model.add(keras.layers.Dense(8))

# Note that you can also omit the initial `Input`.

# In that case the model doesn't have any weights until the first call

# to a training/evaluation method (since it isn't yet built):

model = keras.Sequential()

model.add(keras.layers.Dense(8))

model.add(keras.layers.Dense(4))

# model.weights not created yet

# Whereas if you specify an `Input`, the model gets built

# continuously as you are adding layers:

model = keras.Sequential()

model.add(keras.Input(shape=(16,)))

model.add(keras.layers.Dense(8))

len(model.weights) # Returns "2"

# When using the delayed-build pattern (no input shape specified), you can

# choose to manually build your model by calling

# `build(batch\_input\_shape)`:

model = keras.Sequential()

model.add(keras.layers.Dense(8))

model.add(keras.layers.Dense(4))

model.build((None, 16))

len(model.weights) # Returns "4"

# Note that when using the delayed-build pattern (no input shape specified),

# the model gets built the first time you call `fit`, `eval`, or `predict`,

# or the first time you call the model on some input data.

model = keras.Sequential()

model.add(keras.layers.Dense(8))

model.add(keras.layers.Dense(1))

model.compile(optimizer='sgd', loss='mse')

# This builds the model for the first time:

model.fit(x, y, batch\_size=32, epochs=10)

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/models/sequential.py#L76)

### add method

Sequential.add(layer, rebuild=True)

Adds a layer instance on top of the layer stack.

**Arguments**

* **layer**: layer instance.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/models/sequential.py#L125)

### pop method

Sequential.pop(rebuild=True)

Removes the last layer in the model.

# **Model training APIs**

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/trainers/trainer.py#L29)

### compile method

Model.compile(

optimizer="rmsprop",

loss=None,

loss\_weights=None,

metrics=None,

weighted\_metrics=None,

run\_eagerly=False,

steps\_per\_execution=1,

jit\_compile="auto",

auto\_scale\_loss=True,

)

Configures the model for training.

**Example**

model.compile(

optimizer=keras.optimizers.Adam(learning\_rate=1e-3),

loss=keras.losses.BinaryCrossentropy(),

metrics=[

keras.metrics.BinaryAccuracy(),

keras.metrics.FalseNegatives(),

],

)

**Arguments**

* **optimizer**: String (name of optimizer) or optimizer instance. See keras.optimizers.
* **loss**: Loss function. May be a string (name of loss function), or a keras.losses.Loss instance. See keras.losses. A loss function is any callable with the signature loss = fn(y\_true, y\_pred), where y\_true are the ground truth values, and y\_pred are the model's predictions. y\_true should have shape (batch\_size, d0, .. dN) (except in the case of sparse loss functions such as sparse categorical crossentropy which expects integer arrays of shape (batch\_size, d0, .. dN-1)). y\_pred should have shape (batch\_size, d0, .. dN). The loss function should return a float tensor.
* **loss\_weights**: Optional list or dictionary specifying scalar coefficients (Python floats) to weight the loss contributions of different model outputs. The loss value that will be minimized by the model will then be the weighted sum of all individual losses, weighted by the loss\_weights coefficients. If a list, it is expected to have a 1:1 mapping to the model's outputs. If a dict, it is expected to map output names (strings) to scalar coefficients.
* **metrics**: List of metrics to be evaluated by the model during training and testing. Each of this can be a string (name of a built-in function), function or a [keras.metrics.Metric](https://keras.io/api/metrics/base_metric" \l "metric-class) instance. See keras.metrics. Typically you will use metrics=['accuracy']. A function is any callable with the signature result = fn(y\_true, \_pred). To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as metrics={'a':'accuracy', 'b':['accuracy', 'mse']}. You can also pass a list to specify a metric or a list of metrics for each output, such as metrics=[['accuracy'], ['accuracy', 'mse']] or metrics=['accuracy', ['accuracy', 'mse']]. When you pass the strings 'accuracy' or 'acc', we convert this to one of [keras.metrics.BinaryAccuracy](https://keras.io/api/metrics/accuracy_metrics#binaryaccuracy-class), [keras.metrics.CategoricalAccuracy](https://keras.io/api/metrics/accuracy_metrics#categoricalaccuracy-class), [keras.metrics.SparseCategoricalAccuracy](https://keras.io/api/metrics/accuracy_metrics#sparsecategoricalaccuracy-class) based on the shapes of the targets and of the model output. A similar conversion is done for the strings "crossentropy" and "ce" as well. The metrics passed here are evaluated without sample weighting; if you would like sample weighting to apply, you can specify your metrics via the weighted\_metrics argument instead.
* **weighted\_metrics**: List of metrics to be evaluated and weighted by sample\_weight or class\_weight during training and testing.
* **run\_eagerly**: Bool. If True, this model's forward pass will never be compiled. It is recommended to leave this as False when training (for best performance), and to set it to True when debugging.
* **steps\_per\_execution**: Int. The number of batches to run during each a single compiled function call. Running multiple batches inside a single compiled function call can greatly improve performance on TPUs or small models with a large Python overhead. At most, one full epoch will be run each execution. If a number larger than the size of the epoch is passed, the execution will be truncated to the size of the epoch. Note that if steps\_per\_execution is set to N, Callback.on\_batch\_begin and Callback.on\_batch\_end methods will only be called every N batches (i.e. before/after each compiled function execution). Not supported with the PyTorch backend.
* **jit\_compile**: Bool or "auto". Whether to use XLA compilation when compiling a model. For jax and tensorflow backends, jit\_compile="auto" enables XLA compilation if the model supports it, and disabled otherwise. For torch backend, "auto" will default to eager execution and jit\_compile=True will run with torch.compile with the "inductor" backend.
* **auto\_scale\_loss**: Bool. If True and the model dtype policy is "mixed\_float16", the passed optimizer will be automatically wrapped in a LossScaleOptimizer, which will dynamically scale the loss to prevent underflow.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/backend/tensorflow/trainer.py#L236)

### fit method

Model.fit(

x=None,

y=None,

batch\_size=None,

epochs=1,

verbose="auto",

callbacks=None,

validation\_split=0.0,

validation\_data=None,

shuffle=True,

class\_weight=None,

sample\_weight=None,

initial\_epoch=0,

steps\_per\_epoch=None,

validation\_steps=None,

validation\_batch\_size=None,

validation\_freq=1,

)

Trains the model for a fixed number of epochs (dataset iterations).

**Arguments**

* **x**: Input data. It could be:
  + A NumPy array (or array-like), or a list of arrays (in case the model has multiple inputs).
  + A tensor, or a list of tensors (in case the model has multiple inputs).
  + A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
  + A [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset). Should return a tuple of either (inputs, targets) or (inputs, targets, sample\_weights).
  + A [keras.utils.PyDataset](https://keras.io/api/utils/python_utils" \l "pydataset-class) returning (inputs, targets) or (inputs, targets, sample\_weights).
* **y**: Target data. Like the input data x, it could be either NumPy array(s) or backend-native tensor(s). If x is a dataset, generator, or [keras.utils.PyDataset](https://keras.io/api/utils/python_utils" \l "pydataset-class) instance, y should not be specified (since targets will be obtained from x).
* **batch\_size**: Integer or None. Number of samples per gradient update. If unspecified, batch\_size will default to 32. Do not specify the batch\_size if your data is in the form of datasets, generators, or [keras.utils.PyDataset](https://keras.io/api/utils/python_utils" \l "pydataset-class) instances (since they generate batches).
* **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire x and y data provided (unless the steps\_per\_epoch flag is set to something other than None). Note that in conjunction with initial\_epoch, epochs is to be understood as "final epoch". The model is not trained for a number of iterations given by epochs, but merely until the epoch of index epochs is reached.
* **verbose**: "auto", 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch. "auto" becomes 1 for most cases. Note that the progress bar is not particularly useful when logged to a file, so verbose=2 is recommended when not running interactively (e.g., in a production environment). Defaults to "auto".
* **callbacks**: List of [keras.callbacks.Callback](https://keras.io/api/callbacks/base_callback" \l "callback-class) instances. List of callbacks to apply during training. See keras.callbacks. Note [keras.callbacks.ProgbarLogger](https://keras.io/api/callbacks/progbar_logger#progbarlogger-class) and keras.callbacks.History callbacks are created automatically and need not be passed to model.fit(). [keras.callbacks.ProgbarLogger](https://keras.io/api/callbacks/progbar_logger" \l "progbarlogger-class) is created or not based on the verbose argument in model.fit().
* **validation\_split**: Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data is selected from the last samples in the x and y data provided, before shuffling. This argument is not supported when x is a dataset, generator or [keras.utils.PyDataset](https://keras.io/api/utils/python_utils" \l "pydataset-class) instance. If both validation\_data and validation\_split are provided, validation\_data will override validation\_split.
* **validation\_data**: Data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. Thus, note the fact that the validation loss of data provided using validation\_split or validation\_data is not affected by regularization layers like noise and dropout. validation\_data will override validation\_split. It could be:
  + A tuple (x\_val, y\_val) of NumPy arrays or tensors.
  + A tuple (x\_val, y\_val, val\_sample\_weights) of NumPy arrays.
  + A [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset).
  + A Python generator or [keras.utils.PyDataset](https://keras.io/api/utils/python_utils" \l "pydataset-class) returning (inputs, targets) or (inputs, targets, sample\_weights).
* **shuffle**: Boolean, whether to shuffle the training data before each epoch. This argument is ignored when x is a generator or a [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset).
* **class\_weight**: Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class. When class\_weight is specified and targets have a rank of 2 or greater, either y must be one-hot encoded, or an explicit final dimension of 1 must be included for sparse class labels.
* **sample\_weight**: Optional NumPy array of weights for the training samples, used for weighting the loss function (during training only). You can either pass a flat (1D) NumPy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape (samples, sequence\_length), to apply a different weight to every timestep of every sample. This argument is not supported when x is a dataset, generator, or [keras.utils.PyDataset](https://keras.io/api/utils/python_utils" \l "pydataset-class) instance, instead provide the sample\_weights as the third element of x. Note that sample weighting does not apply to metrics specified via the metrics argument in compile(). To apply sample weighting to your metrics, you can specify them via the weighted\_metrics in compile() instead.
* **initial\_epoch**: Integer. Epoch at which to start training (useful for resuming a previous training run).
* **steps\_per\_epoch**: Integer or None. Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. When training with input tensors such as backend-native tensors, the default None is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined. If x is a [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset), and steps\_per\_epoch is None, the epoch will run until the input dataset is exhausted. When passing an infinitely repeating dataset, you must specify the steps\_per\_epoch argument. If steps\_per\_epoch=-1 the training will run indefinitely with an infinitely repeating dataset.
* **validation\_steps**: Only relevant if validation\_data is provided. Total number of steps (batches of samples) to draw before stopping when performing validation at the end of every epoch. If validation\_steps is None, validation will run until the validation\_data dataset is exhausted. In the case of an infinitely repeated dataset, it will run into an infinite loop. If validation\_steps is specified and only part of the dataset will be consumed, the evaluation will start from the beginning of the dataset at each epoch. This ensures that the same validation samples are used every time.
* **validation\_batch\_size**: Integer or None. Number of samples per validation batch. If unspecified, will default to batch\_size. Do not specify the validation\_batch\_size if your data is in the form of datasets or [keras.utils.PyDataset](https://keras.io/api/utils/python_utils" \l "pydataset-class) instances (since they generate batches).
* **validation\_freq**: Only relevant if validation data is provided. Specifies how many training epochs to run before a new validation run is performed, e.g. validation\_freq=2 runs validation every 2 epochs.

Unpacking behavior for iterator-like inputs: A common pattern is to pass an iterator like object such as a [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset) or a [keras.utils.PyDataset](https://keras.io/api/utils/python_utils" \l "pydataset-class) to fit(), which will in fact yield not only features (x) but optionally targets (y) and sample weights (sample\_weight). Keras requires that the output of such iterator-likes be unambiguous. The iterator should return a tuple of length 1, 2, or 3, where the optional second and third elements will be used for y and sample\_weight respectively. Any other type provided will be wrapped in a length-one tuple, effectively treating everything as x. When yielding dicts, they should still adhere to the top-level tuple structure, e.g. ({"x0": x0, "x1": x1}, y). Keras will not attempt to separate features, targets, and weights from the keys of a single dict. A notable unsupported data type is the namedtuple. The reason is that it behaves like both an ordered datatype (tuple) and a mapping datatype (dict). So given a namedtuple of the form: namedtuple("example\_tuple", ["y", "x"]) it is ambiguous whether to reverse the order of the elements when interpreting the value. Even worse is a tuple of the form: namedtuple("other\_tuple", ["x", "y", "z"]) where it is unclear if the tuple was intended to be unpacked into x, y, and sample\_weight or passed through as a single element to x.

**Returns**

A History object. Its History.history attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/backend/tensorflow/trainer.py#L371)

### evaluate method

Model.evaluate(

x=None,

y=None,

batch\_size=None,

verbose="auto",

sample\_weight=None,

steps=None,

callbacks=None,

return\_dict=False,

\*\*kwargs

)

Returns the loss value & metrics values for the model in test mode.

Computation is done in batches (see the batch\_size arg.)

**Arguments**

* **x**: Input data. It could be:
  + A NumPy array (or array-like), or a list of arrays (in case the model has multiple inputs).
  + A tensor, or a list of tensors (in case the model has multiple inputs).
  + A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
  + A [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset). Should return a tuple of either (inputs, targets) or (inputs, targets, sample\_weights).
  + A generator or [keras.utils.PyDataset](https://keras.io/api/utils/python_utils" \l "pydataset-class) returning (inputs, targets) or (inputs, targets, sample\_weights).
* **y**: Target data. Like the input data x, it could be either NumPy array(s) or backend-native tensor(s). If x is a [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset) or [keras.utils.PyDataset](https://keras.io/api/utils/python_utils" \l "pydataset-class) instance, y should not be specified (since targets will be obtained from the iterator/dataset).
* **batch\_size**: Integer or None. Number of samples per batch of computation. If unspecified, batch\_size will default to 32. Do not specify the batch\_size if your data is in the form of a dataset, generators, or [keras.utils.PyDataset](https://keras.io/api/utils/python_utils" \l "pydataset-class) instances (since they generate batches).
* **verbose**: "auto", 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = single line. "auto" becomes 1 for most cases. Note that the progress bar is not particularly useful when logged to a file, so verbose=2 is recommended when not running interactively (e.g. in a production environment). Defaults to "auto".
* **sample\_weight**: Optional NumPy array of weights for the test samples, used for weighting the loss function. You can either pass a flat (1D) NumPy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape (samples, sequence\_length), to apply a different weight to every timestep of every sample. This argument is not supported when x is a dataset, instead pass sample weights as the third element of x.
* **steps**: Integer or None. Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of None. If x is a [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset) and steps is None, evaluation will run until the dataset is exhausted.
* **callbacks**: List of [keras.callbacks.Callback](https://keras.io/api/callbacks/base_callback" \l "callback-class) instances. List of callbacks to apply during evaluation.
* **return\_dict**: If True, loss and metric results are returned as a dict, with each key being the name of the metric. If False, they are returned as a list.

**Returns**

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute model.metrics\_names will give you the display labels for the scalar outputs.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/backend/tensorflow/trainer.py#L437)

### predict method

Model.predict(x, batch\_size=None, verbose="auto", steps=None, callbacks=None)

Generates output predictions for the input samples.

Computation is done in batches. This method is designed for batch processing of large numbers of inputs. It is not intended for use inside of loops that iterate over your data and process small numbers of inputs at a time.

For small numbers of inputs that fit in one batch, directly use \_\_call\_\_() for faster execution, e.g., model(x), or model(x, training=False) if you have layers such as BatchNormalization that behave differently during inference.

Note: See [this FAQ entry](https://keras.io/getting_started/faq/#whats-the-difference-between-model-methods-predict-and-call) for more details about the difference between Model methods predict() and \_\_call\_\_().

**Arguments**

* **x**: Input samples. It could be:
  + A NumPy array (or array-like), or a list of arrays (in case the model has multiple inputs).
  + A tensor, or a list of tensors (in case the model has multiple inputs).
  + A [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset).
  + A [keras.utils.PyDataset](https://keras.io/api/utils/python_utils" \l "pydataset-class) instance.
* **batch\_size**: Integer or None. Number of samples per batch. If unspecified, batch\_size will default to 32. Do not specify the batch\_size if your data is in the form of dataset, generators, or [keras.utils.PyDataset](https://keras.io/api/utils/python_utils" \l "pydataset-class) instances (since they generate batches).
* **verbose**: "auto", 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = single line. "auto" becomes 1 for most cases. Note that the progress bar is not particularly useful when logged to a file, so verbose=2 is recommended when not running interactively (e.g. in a production environment). Defaults to "auto".
* **steps**: Total number of steps (batches of samples) before declaring the prediction round finished. Ignored with the default value of None. If x is a [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset) and steps is None, predict() will run until the input dataset is exhausted.
* **callbacks**: List of [keras.callbacks.Callback](https://keras.io/api/callbacks/base_callback" \l "callback-class) instances. List of callbacks to apply during prediction.

**Returns**

NumPy array(s) of predictions.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/backend/tensorflow/trainer.py#L515)

### train\_on\_batch method

Model.train\_on\_batch(

x, y=None, sample\_weight=None, class\_weight=None, return\_dict=False

)

Runs a single gradient update on a single batch of data.

**Arguments**

* **x**: Input data. Must be array-like.
* **y**: Target data. Must be array-like.
* **sample\_weight**: Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence\_length), to apply a different weight to every timestep of every sample.
* **class\_weight**: Optional dictionary mapping class indices (integers) to a weight (float) to apply to the model's loss for the samples from this class during training. This can be useful to tell the model to "pay more attention" to samples from an under-represented class. When class\_weight is specified and targets have a rank of 2 or greater, either y must be one-hot encoded, or an explicit final dimension of 1 must be included for sparse class labels.
* **return\_dict**: If True, loss and metric results are returned as a dict, with each key being the name of the metric. If False, they are returned as a list.

**Returns**

A scalar loss value (when no metrics and return\_dict=False), a list of loss and metric values (if there are metrics and return\_dict=False), or a dict of metric and loss values (if return\_dict=True).

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/backend/tensorflow/trainer.py#L546)

### test\_on\_batch method

Model.test\_on\_batch(x, y=None, sample\_weight=None, return\_dict=False)

Test the model on a single batch of samples.

**Arguments**

* **x**: Input data. Must be array-like.
* **y**: Target data. Must be array-like.
* **sample\_weight**: Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence\_length), to apply a different weight to every timestep of every sample.
* **return\_dict**: If True, loss and metric results are returned as a dict, with each key being the name of the metric. If False, they are returned as a list.

**Returns**

A scalar loss value (when no metrics and return\_dict=False), a list of loss and metric values (if there are metrics and return\_dict=False), or a dict of metric and loss values (if return\_dict=True).

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/backend/tensorflow/trainer.py#L565)

### predict\_on\_batch method

Model.predict\_on\_batch(x)

Returns predictions for a single batch of samples.

**Arguments**

* **x**: Input data. It must be array-like.

**Returns**

NumPy array(s) of predictions.

# **Saving & serialization**

# **Whole model saving & loading**

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/models/model.py#L266)

### save method

Model.save(filepath, overwrite=True, \*\*kwargs)

Saves a model as a .keras file.

**Arguments**

* **filepath**: str or pathlib.Path object. Path where to save the model. Must end in .keras.
* **overwrite**: Whether we should overwrite any existing model at the target location, or instead ask the user via an interactive prompt.
* **save\_format**: The save\_format argument is deprecated in Keras 3. Format to use, as a string. Only the "keras" format is supported at this time.

**Example**

model = keras.Sequential(

[

keras.layers.Dense(5, input\_shape=(3,)),

keras.layers.Softmax(),

],

)

model.save("model.keras")

loaded\_model = keras.saving.load\_model("model.keras")

x = keras.random.uniform((10, 3))

assert np.allclose(model.predict(x), loaded\_model.predict(x))

Note that model.save() is an alias for keras.saving.save\_model().

The saved .keras file contains:

* The model's configuration (architecture)
* The model's weights
* The model's optimizer's state (if any)

Thus models can be reinstantiated in the exact same state.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/saving/saving_api.py#L18)

### save\_model function

keras.saving.save\_model(model, filepath, overwrite=True, \*\*kwargs)

Saves a model as a .keras file.

**Arguments**

* **model**: Keras model instance to be saved.
* **filepath**: str or pathlib.Path object. Path where to save the model.
* **overwrite**: Whether we should overwrite any existing model at the target location, or instead ask the user via an interactive prompt.

**Example**

model = keras.Sequential(

[

keras.layers.Dense(5, input\_shape=(3,)),

keras.layers.Softmax(),

],

)

model.save("model.keras")

loaded\_model = keras.saving.load\_model("model.keras")

x = keras.random.uniform((10, 3))

assert np.allclose(model.predict(x), loaded\_model.predict(x))

Note that model.save() is an alias for keras.saving.save\_model().

The saved .keras file contains:

* The model's configuration (architecture)
* The model's weights
* The model's optimizer's state (if any)

Thus models can be reinstantiated in the exact same state.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/saving/saving_api.py#L116)

### load\_model function

keras.saving.load\_model(filepath, custom\_objects=None, compile=True, safe\_mode=True)

Loads a model saved via model.save().

**Arguments**

* **filepath**: str or pathlib.Path object, path to the saved model file.
* **custom\_objects**: Optional dictionary mapping names (strings) to custom classes or functions to be considered during deserialization.
* **compile**: Boolean, whether to compile the model after loading.
* **safe\_mode**: Boolean, whether to disallow unsafe lambda deserialization. When safe\_mode=False, loading an object has the potential to trigger arbitrary code execution. This argument is only applicable to the Keras v3 model format. Defaults to True.

**Returns**

A Keras model instance. If the original model was compiled, and the argument compile=True is set, then the returned model will be compiled. Otherwise, the model will be left uncompiled.

**Example**

model = keras.Sequential([

keras.layers.Dense(5, input\_shape=(3,)),

keras.layers.Softmax()])

model.save("model.keras")

loaded\_model = keras.saving.load\_model("model.keras")

x = np.random.random((10, 3))

assert np.allclose(model.predict(x), loaded\_model.predict(x))

Note that the model variables may have different name values (var.name property, e.g. "dense\_1/kernel:0") after being reloaded. It is recommended that you use layer attributes to access specific variables, e.g. model.get\_layer("dense\_1").kernel.

# **Weights-only saving & loading**

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/models/model.py#L307)

### save\_weights method

Model.save\_weights(filepath, overwrite=True)

Saves all layer weights to a .weights.h5 file.

**Arguments**

* **filepath**: str or pathlib.Path object. Path where to save the model. Must end in .weights.h5.
* **overwrite**: Whether we should overwrite any existing model at the target location, or instead ask the user via an interactive prompt.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/models/model.py#L320)

### load\_weights method

Model.load\_weights(filepath, skip\_mismatch=False, \*\*kwargs)

Load weights from a file saved via save\_weights().

Weights are loaded based on the network's topology. This means the architecture should be the same as when the weights were saved. Note that layers that don't have weights are not taken into account in the topological ordering, so adding or removing layers is fine as long as they don't have weights.

**Partial weight loading**

If you have modified your model, for instance by adding a new layer (with weights) or by changing the shape of the weights of a layer, you can choose to ignore errors and continue loading by setting skip\_mismatch=True. In this case any layer with mismatching weights will be skipped. A warning will be displayed for each skipped layer.

**Arguments**

* **filepath**: String, path to the weights file to load. It can either be a .weights.h5 file or a legacy .h5 weights file.
* **skip\_mismatch**: Boolean, whether to skip loading of layers where there is a mismatch in the number of weights, or a mismatch in the shape of the weights.

# **Model config serialization**

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/layers/layer.py#L1435)

### get\_config method

Model.get\_config()

Returns the config of the object.

An object config is a Python dictionary (serializable) containing the information needed to re-instantiate it.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/models/model.py#L490)

### from\_config method

Model.from\_config(config, custom\_objects=None)

Creates a layer from its config.

This method is the reverse of get\_config, capable of instantiating the same layer from the config dictionary. It does not handle layer connectivity (handled by Network), nor weights (handled by set\_weights).

**Arguments**

* **config**: A Python dictionary, typically the output of get\_config.

**Returns**

A layer instance.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/models/cloning.py#L13)

### clone\_model function

keras.models.clone\_model(

model,

input\_tensors=None,

clone\_function=None,

call\_function=None,

recursive=False,

\*\*kwargs

)

Clone a Functional or Sequential Model instance.

Model cloning is similar to calling a model on new inputs, except that it creates new layers (and thus new weights) instead of sharing the weights of the existing layers.

Note that clone\_model will not preserve the uniqueness of shared objects within the model (e.g. a single variable attached to two distinct layers will be restored as two separate variables).

**Arguments**

* **model**: Instance of Model (could be a Functional model or a Sequential model).
* **input\_tensors**: optional list of input tensors or InputLayer objects to build the model upon. If not provided, new Input objects will be created.
* **clone\_function**: Callable with signature fn(layer) to be used to clone each layer in the target model (except Input instances). It takes as argument the layer instance to be cloned, and returns the corresponding layer instance to be used in the model copy. If unspecified, this callable defaults to the following serialization/deserialization function: lambda layer: layer.\_\_class\_\_.from\_config(layer.get\_config()). By passing a custom callable, you can customize your copy of the model, e.g. by wrapping certain layers of interest (you might want to replace all LSTM instances with equivalent Bidirectional(LSTM(...)) instances, for example). Defaults to None.
* **call\_function**: Callable with signature fn(layer, \*args, \*\*kwargs) to be used to call each cloned layer and a set of inputs. It takes the layer instance, the call arguments and keyword arguments, and returns the call outputs. If unspecified, this callable defaults to the regular \_\_call\_\_() method: def fn(layer, \*args, \*\*kwargs): return layer(\*args, \*\*kwargs). By passing a custom callable, you can insert new layers before or after a given layer. Note: this argument can only be used with Functional models.
* **recursive**: Boolean. Whether to recursively clone any Sequential or Functional models encountered in the original Sequential/Functional model. If False, then inner models are cloned by calling clone\_function(). If True, then inner models are cloned by calling clone\_model() with the same clone\_function, call\_function, and recursive arguments. Note that in this case, call\_function will not be propagated to any Sequential model (since it is not applicable to Sequential models).

**Returns**

An instance of Model reproducing the behavior of the original model, on top of new inputs tensors, using newly instantiated weights. The cloned model may behave differently from the original model if a custom clone\_function or call\_function modifies a layer or layer call.

**Example**

# Create a test Sequential model.

model = keras.Sequential([

keras.layers.Input(shape=(728,)),

keras.layers.Dense(32, activation='relu'),

keras.layers.Dense(1, activation='sigmoid'),

])

# Create a copy of the test model (with freshly initialized weights).

new\_model = clone\_model(model)

Using a clone\_function to make a model deterministic by setting the random seed everywhere:

def clone\_function(layer):

config = layer.get\_config()

if "seed" in config:

config["seed"] = 1337

return layer.\_\_class\_\_.from\_config(config)

new\_model = clone\_model(model)

Using a call\_function to add a Dropout layer after each Dense layer (without recreating new layers):

def call\_function(layer, \*args, \*\*kwargs):

out = layer(\*args, \*\*kwargs)

if isinstance(layer, keras.layers.Dense):

out = keras.layers.Dropout(0.5)(out)

return out

new\_model = clone\_model(

model,

clone\_function=lambda x: x, # Reuse the same layers.

call\_function=call\_function,

)

Note that subclassed models cannot be cloned by default, since their internal layer structure is not known. To achieve equivalent functionality as clone\_model in the case of a subclassed model, simply make sure that the model class implements get\_config() (and optionally from\_config()), and call:

new\_model = model.\_\_class\_\_.from\_config(model.get\_config())

In the case of a subclassed model, you cannot using a custom clone\_function.

# **Model export for inference**

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/models/model.py#L452)

### export method

Model.export(filepath, format="tf\_saved\_model")

Create a TF SavedModel artifact for inference.

**Note:** This can currently only be used with the TensorFlow or JAX backends.

This method lets you export a model to a lightweight SavedModel artifact that contains the model's forward pass only (its call() method) and can be served via e.g. TF-Serving. The forward pass is registered under the name serve() (see example below).

The original code of the model (including any custom layers you may have used) is no longer necessary to reload the artifact – it is entirely standalone.

**Arguments**

* **filepath**: str or pathlib.Path object. Path where to save the artifact.

**Example**

# Create the artifact

model.export("path/to/location")

# Later, in a different process / environment...

reloaded\_artifact = tf.saved\_model.load("path/to/location")

predictions = reloaded\_artifact.serve(input\_data)

If you would like to customize your serving endpoints, you can use the lower-level [keras.export.ExportArchive](https://keras.io/api/models/model_saving_apis/export" \l "exportarchive-class) class. The export() method relies on ExportArchive internally.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/export/export_lib.py#L20)

### ExportArchive class

keras.export.ExportArchive()

ExportArchive is used to write SavedModel artifacts (e.g. for inference).

If you have a Keras model or layer that you want to export as SavedModel for serving (e.g. via TensorFlow-Serving), you can use ExportArchive to configure the different serving endpoints you need to make available, as well as their signatures. Simply instantiate an ExportArchive, use track() to register the layer(s) or model(s) to be used, then use the add\_endpoint() method to register a new serving endpoint. When done, use the write\_out() method to save the artifact.

The resulting artifact is a SavedModel and can be reloaded via [tf.saved\_model.load](https://www.tensorflow.org/api_docs/python/tf/saved_model/load).

**Examples**

Here's how to export a model for inference.

export\_archive = ExportArchive()

export\_archive.track(model)

export\_archive.add\_endpoint(

name="serve",

fn=model.call,

input\_signature=[tf.TensorSpec(shape=(None, 3), dtype=tf.float32)],

)

export\_archive.write\_out("path/to/location")

# Elsewhere, we can reload the artifact and serve it.

# The endpoint we added is available as a method:

serving\_model = tf.saved\_model.load("path/to/location")

outputs = serving\_model.serve(inputs)

Here's how to export a model with one endpoint for inference and one endpoint for a training-mode forward pass (e.g. with dropout on).

export\_archive = ExportArchive()

export\_archive.track(model)

export\_archive.add\_endpoint(

name="call\_inference",

fn=lambda x: model.call(x, training=False),

input\_signature=[tf.TensorSpec(shape=(None, 3), dtype=tf.float32)],

)

export\_archive.add\_endpoint(

name="call\_training",

fn=lambda x: model.call(x, training=True),

input\_signature=[tf.TensorSpec(shape=(None, 3), dtype=tf.float32)],

)

export\_archive.write\_out("path/to/location")

**Note on resource tracking:**

ExportArchive is able to automatically track all [tf.Variables](https://www.tensorflow.org/api_docs/python/tf/Variables) used by its endpoints, so most of the time calling .track(model) is not strictly required. However, if your model uses lookup layers such as IntegerLookup, StringLookup, or TextVectorization, it will need to be tracked explicitly via .track(model).

Explicit tracking is also required if you need to be able to access the properties variables, trainable\_variables, or non\_trainable\_variables on the revived archive.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/export/export_lib.py#L200)

### add\_endpoint method

ExportArchive.add\_endpoint(name, fn, input\_signature=None, jax2tf\_kwargs=None)

Register a new serving endpoint.

**Arguments**

* **name**: Str, name of the endpoint.
* **fn**: A function. It should only leverage resources (e.g. [tf.Variable](https://www.tensorflow.org/api_docs/python/tf/Variable) objects or [tf.lookup.StaticHashTable](https://www.tensorflow.org/api_docs/python/tf/lookup/StaticHashTable) objects) that are available on the models/layers tracked by the ExportArchive (you can call .track(model) to track a new model). The shape and dtype of the inputs to the function must be known. For that purpose, you can either 1) make sure that fn is a [tf.function](https://www.tensorflow.org/api_docs/python/tf/function) that has been called at least once, or 2) provide an input\_signature argument that specifies the shape and dtype of the inputs (see below).
* **input\_signature**: Used to specify the shape and dtype of the inputs to fn. List of [tf.TensorSpec](https://www.tensorflow.org/api_docs/python/tf/TensorSpec) objects (one per positional input argument of fn). Nested arguments are allowed (see below for an example showing a Functional model with 2 input arguments).
* **jax2tf\_kwargs**: Optional. A dict for arguments to pass to jax2tf. Supported only when the backend is JAX. See documentation for [jax2tf.convert](https://github.com/google/jax/blob/main/jax/experimental/jax2tf/README.md). The values for native\_serialization and polymorphic\_shapes, if not provided, are automatically computed.

**Returns**

The [tf.function](https://www.tensorflow.org/api_docs/python/tf/function) wrapping fn that was added to the archive.

**Example**

Adding an endpoint using the input\_signature argument when the model has a single input argument:

export\_archive = ExportArchive()

export\_archive.track(model)

export\_archive.add\_endpoint(

name="serve",

fn=model.call,

input\_signature=[tf.TensorSpec(shape=(None, 3), dtype=tf.float32)],

)

Adding an endpoint using the input\_signature argument when the model has two positional input arguments:

export\_archive = ExportArchive()

export\_archive.track(model)

export\_archive.add\_endpoint(

name="serve",

fn=model.call,

input\_signature=[

tf.TensorSpec(shape=(None, 3), dtype=tf.float32),

tf.TensorSpec(shape=(None, 4), dtype=tf.float32),

],

)

Adding an endpoint using the input\_signature argument when the model has one input argument that is a list of 2 tensors (e.g. a Functional model with 2 inputs):

model = keras.Model(inputs=[x1, x2], outputs=outputs)

export\_archive = ExportArchive()

export\_archive.track(model)

export\_archive.add\_endpoint(

name="serve",

fn=model.call,

input\_signature=[

[

tf.TensorSpec(shape=(None, 3), dtype=tf.float32),

tf.TensorSpec(shape=(None, 4), dtype=tf.float32),

],

],

)

This also works with dictionary inputs:

model = keras.Model(inputs={"x1": x1, "x2": x2}, outputs=outputs)

export\_archive = ExportArchive()

export\_archive.track(model)

export\_archive.add\_endpoint(

name="serve",

fn=model.call,

input\_signature=[

{

"x1": tf.TensorSpec(shape=(None, 3), dtype=tf.float32),

"x2": tf.TensorSpec(shape=(None, 4), dtype=tf.float32),

},

],

)

Adding an endpoint that is a [tf.function](https://www.tensorflow.org/api_docs/python/tf/function):

@tf.function()

def serving\_fn(x):

return model(x)

# The function must be traced, i.e. it must be called at least once.

serving\_fn(tf.random.normal(shape=(2, 3)))

export\_archive = ExportArchive()

export\_archive.track(model)

export\_archive.add\_endpoint(name="serve", fn=serving\_fn)

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/export/export_lib.py#L405)

### add\_variable\_collection method

ExportArchive.add\_variable\_collection(name, variables)

Register a set of variables to be retrieved after reloading.

**Arguments**

* **name**: The string name for the collection.
* **variables**: A tuple/list/set of [tf.Variable](https://www.tensorflow.org/api_docs/python/tf/Variable) instances.

**Example**

export\_archive = ExportArchive()

export\_archive.track(model)

# Register an endpoint

export\_archive.add\_endpoint(

name="serve",

fn=model.call,

input\_signature=[tf.TensorSpec(shape=(None, 3), dtype=tf.float32)],

)

# Save a variable collection

export\_archive.add\_variable\_collection(

name="optimizer\_variables", variables=model.optimizer.variables)

export\_archive.write\_out("path/to/location")

# Reload the object

revived\_object = tf.saved\_model.load("path/to/location")

# Retrieve the variables

optimizer\_variables = revived\_object.optimizer\_variables

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/export/export_lib.py#L119)

### track method

ExportArchive.track(resource)

Track the variables (and other assets) of a layer or model.

By default, all variables used by an endpoint function are automatically tracked when you call add\_endpoint(). However, non-variables assets such as lookup tables need to be tracked manually. Note that lookup tables used by built-in Keras layers (TextVectorization, IntegerLookup, StringLookup) are automatically tracked in add\_endpoint().

**Arguments**

* **resource**: A trackable TensorFlow resource.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/export/export_lib.py#L453)

### write\_out method

ExportArchive.write\_out(filepath, options=None)

Write the corresponding SavedModel to disk.

**Arguments**

* **filepath**: str or pathlib.Path object. Path where to save the artifact.
* **options**: [tf.saved\_model.SaveOptions](https://www.tensorflow.org/api_docs/python/tf/saved_model/SaveOptions) object that specifies SavedModel saving options.

**Note on TF-Serving**: all endpoints registered via add\_endpoint() are made visible for TF-Serving in the SavedModel artifact. In addition, the first endpoint registered is made visible under the alias "serving\_default" (unless an endpoint with the name "serving\_default" was already registered manually), since TF-Serving requires this endpoint to be set.

# **Serialization utilities**

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/saving/serialization_lib.py#L117)

### serialize\_keras\_object function

keras.saving.serialize\_keras\_object(obj)

Retrieve the config dict by serializing the Keras object.

serialize\_keras\_object() serializes a Keras object to a python dictionary that represents the object, and is a reciprocal function of deserialize\_keras\_object(). See deserialize\_keras\_object() for more information about the config format.

**Arguments**

* **obj**: the Keras object to serialize.

**Returns**

A python dict that represents the object. The python dict can be deserialized via deserialize\_keras\_object().

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/saving/serialization_lib.py#L393)

### deserialize\_keras\_object function

keras.saving.deserialize\_keras\_object(

config, custom\_objects=None, safe\_mode=True, \*\*kwargs

)

Retrieve the object by deserializing the config dict.

The config dict is a Python dictionary that consists of a set of key-value pairs, and represents a Keras object, such as an Optimizer, Layer, Metrics, etc. The saving and loading library uses the following keys to record information of a Keras object:

* class\_name: String. This is the name of the class, as exactly defined in the source code, such as "LossesContainer".
* config: Dict. Library-defined or user-defined key-value pairs that store the configuration of the object, as obtained by object.get\_config().
* module: String. The path of the python module. Built-in Keras classes expect to have prefix keras.
* registered\_name: String. The key the class is registered under via keras.saving.register\_keras\_serializable(package, name) API. The key has the format of '{package}>{name}', where package and name are the arguments passed to register\_keras\_serializable(). If name is not provided, it uses the class name. If registered\_name successfully resolves to a class (that was registered), the class\_name and config values in the dict will not be used. registered\_name is only used for non-built-in classes.

For example, the following dictionary represents the built-in Adam optimizer with the relevant config:

dict\_structure = {

"class\_name": "Adam",

"config": {

"amsgrad": false,

"beta\_1": 0.8999999761581421,

"beta\_2": 0.9990000128746033,

"decay": 0.0,

"epsilon": 1e-07,

"learning\_rate": 0.0010000000474974513,

"name": "Adam"

},

"module": "keras.optimizers",

"registered\_name": None

}

# Returns an `Adam` instance identical to the original one.

deserialize\_keras\_object(dict\_structure)

If the class does not have an exported Keras namespace, the library tracks it by its module and class\_name. For example:

dict\_structure = {

"class\_name": "MetricsList",

"config": {

...

},

"module": "keras.trainers.compile\_utils",

"registered\_name": "MetricsList"

}

# Returns a `MetricsList` instance identical to the original one.

deserialize\_keras\_object(dict\_structure)

And the following dictionary represents a user-customized MeanSquaredError loss:

@keras.saving.register\_keras\_serializable(package='my\_package')

class ModifiedMeanSquaredError(keras.losses.MeanSquaredError):

...

dict\_structure = {

"class\_name": "ModifiedMeanSquaredError",

"config": {

"fn": "mean\_squared\_error",

"name": "mean\_squared\_error",

"reduction": "auto"

},

"registered\_name": "my\_package>ModifiedMeanSquaredError"

}

# Returns the `ModifiedMeanSquaredError` object

deserialize\_keras\_object(dict\_structure)

**Arguments**

* **config**: Python dict describing the object.
* **custom\_objects**: Python dict containing a mapping between custom object names the corresponding classes or functions.
* **safe\_mode**: Boolean, whether to disallow unsafe lambda deserialization. When safe\_mode=False, loading an object has the potential to trigger arbitrary code execution. This argument is only applicable to the Keras v3 model format. Defaults to True.

**Returns**

The object described by the config dictionary.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/saving/object_registration.py#L10)

### CustomObjectScope class

keras.saving.custom\_object\_scope(custom\_objects)

Exposes custom classes/functions to Keras deserialization internals.

Under a scope with custom\_object\_scope(objects\_dict), Keras methods such as keras.models.load\_model() or keras.models.model\_from\_config() will be able to deserialize any custom object referenced by a saved config (e.g. a custom layer or metric).

**Example**

Consider a custom regularizer my\_regularizer:

layer = Dense(3, kernel\_regularizer=my\_regularizer)

# Config contains a reference to `my\_regularizer`

config = layer.get\_config()

...

# Later:

with custom\_object\_scope({'my\_regularizer': my\_regularizer}):

layer = Dense.from\_config(config)

**Arguments**

* **custom\_objects**: Dictionary of {str: object} pairs, where the str key is the object name.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/saving/object_registration.py#L68)

### get\_custom\_objects function

keras.saving.get\_custom\_objects()

Retrieves a live reference to the global dictionary of custom objects.

Custom objects set using custom\_object\_scope() are not added to the global dictionary of custom objects, and will not appear in the returned dictionary.

**Example**

get\_custom\_objects().clear()

get\_custom\_objects()['MyObject'] = MyObject

**Returns**

Global dictionary mapping registered class names to classes.

[[source]](https://github.com/keras-team/keras/tree/v3.3.3/keras/src/saving/object_registration.py#L94)

### register\_keras\_serializable function

keras.saving.register\_keras\_serializable(package="Custom", name=None)

Registers an object with the Keras serialization framework.

This decorator injects the decorated class or function into the Keras custom object dictionary, so that it can be serialized and deserialized without needing an entry in the user-provided custom object dict. It also injects a function that Keras will call to get the object's serializable string key.

Note that to be serialized and deserialized, classes must implement the get\_config() method. Functions do not have this requirement.

The object will be registered under the key 'package>name' where name, defaults to the object name if not passed.

**Example**

# Note that `'my\_package'` is used as the `package` argument here, and since

# the `name` argument is not provided, `'MyDense'` is used as the `name`.

@register\_keras\_serializable('my\_package')

class MyDense(keras.layers.Dense):

pass

assert get\_registered\_object('my\_package>MyDense') == MyDense

assert get\_registered\_name(MyDense) == 'my\_package>MyDense'

**Arguments**

* **package**: The package that this class belongs to. This is used for the key (which is "package>name") to idenfify the class. Note that this is the first argument passed into the decorator.
* **name**: The name to serialize this class under in this package. If not provided or None, the class' name will be used (note that this is the case when the decorator is used with only one argument, which becomes the package).

**Returns**

A decorator that registers the decorated class with the passed names.

# **Keras layers API**

Layers are the basic building blocks of neural networks in Keras. A layer consists of a tensor-in tensor-out computation function (the layer's call method) and some state, held in TensorFlow variables (the layer's weights).

A Layer instance is callable, much like a function:

import keras

from keras import layers

layer = layers.Dense(32, activation='relu')

inputs = keras.random.uniform(shape=(10, 20))

outputs = layer(inputs)

Unlike a function, though, layers maintain a state, updated when the layer receives data during training, and stored in layer.weights:

>>> layer.weights

[<KerasVariable shape=(20, 32), dtype=float32, path=dense/kernel>,

<KerasVariable shape=(32,), dtype=float32, path=dense/bias>]

## Creating custom layers

While Keras offers a wide range of built-in layers, they don't cover ever possible use case. Creating custom layers is very common, and very easy.

See the guide [Making new layers and models via subclassing](https://keras.io/guides/making_new_layers_and_models_via_subclassing) for an extensive overview, and refer to the documentation for [the base Layer class](https://keras.io/api/layers/base_layer).

## **Layers API overview**