

Model class API

In the functional API, given some input tensor(s) and output tensor(s), you can instantiate a

`Model` via:

```
from keras.models import Model
from keras.layers import Input, Dense

a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(inputs=a, outputs=b)
```

This model will include all layers required in the computation of `b` given `a`.

In the case of multi-input or multi-output models, you can use lists as well:

```
model = Model(inputs=[a1, a2], outputs=[b1, b2, b3])
```

For a detailed introduction of what `Model` can do, read [this guide to the Keras functional API](#).

Methods

compile

```
compile(optimizer, loss=None, metrics=None, loss_weights=None, sample_weight_mode=None, weighted_metrics=None)
```

Configures the model for training.

Arguments

- **optimizer**: String (name of optimizer) or optimizer instance. See [optimizers](#).
- **loss**: String (name of objective function) or objective function. See [losses](#). If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of losses. The loss value that will be minimized by the model will then be the sum of all individual losses.

- **metrics:** List of metrics to be evaluated by the model during training and testing. Typically you will use `metrics=['accuracy']`. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as `metrics={'output_a': 'accuracy'}`.
- **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 tensor, it is expected to map output names (strings) to scalar coefficients.
- **sample_weight_mode:** If you need to do timestep-wise sample weighting (2D weights), set this to `"temporal"`. `None` defaults to sample-wise weights (1D). If the model has multiple outputs, you can use a different `sample_weight_mode` on each output by passing a dictionary or a list of modes.
- **weighted_metrics:** List of metrics to be evaluated and weighted by `sample_weight` or `class_weight` during training and testing.
- **target_tensors:** By default, Keras will create placeholders for the model's target, which will be fed with the target data during training. If instead you would like to use your own target tensors (in turn, Keras will not expect external Numpy data for these targets at training time), you can specify them via the `target_tensors` argument. It can be a single tensor (for a single-output model), a list of tensors, or a dict mapping output names to target tensors.
- ****kwargs:** When using the Theano/CNTK backends, these arguments are passed into `K.function`. When using the TensorFlow backend, these arguments are passed into `tf.Session.run`.

Raises

- **ValueError:** In case of invalid arguments for `optimizer`, `loss`, `metrics` OR `sample_weight_mode`.

fit

```
fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_split=0.0, valid
```

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

Arguments

- **x:** Numpy array of training data (if the model has a single input), or list of Numpy arrays (if the model has multiple inputs). If input layers in the model are named, you can also pass a dictionary mapping input names to Numpy arrays. `x` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).

- **y**: Numpy array of target (label) data (if the model has a single output), or list of Numpy arrays (if the model has multiple outputs). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays. `y` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **batch_size**: Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32.
- **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided. 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**: Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during training and validation (if). See [callbacks](#).
- **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.
- **validation_data**: tuple `(x_val, y_val)` or tuple `(x_val, y_val, val_sample_weights)` 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. `validation_data` will override `validation_split`.
- **shuffle**: Boolean (whether to shuffle the training data before each epoch) or str (for 'batch'). 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when `steps_per_epoch` is not `None`.
- **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.
- **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. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.
- **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 TensorFlow data 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.
- **validation_steps**: Only relevant if `steps_per_epoch` is specified. Total number of steps (batches of samples) to validate before stopping.

- **validation_freq**: Only relevant if validation data is provided. Integer or list/tuple/set. If an integer, specifies how many training epochs to run before a new validation run is performed, e.g. `validation_freq=2` runs validation every 2 epochs. If a list, tuple, or set, specifies the epochs on which to run validation, e.g. `validation_freq=[1, 2, 10]` runs validation at the end of the 1st, 2nd, and 10th epochs.

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).

Raises

- **RuntimeError**: If the model was never compiled.
- **ValueError**: In case of mismatch between the provided input data and what the model expects.

evaluate

```
evaluate(x=None, y=None, batch_size=None, verbose=1, sample_weight=None, steps=None, callbacks=None)
```

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

Computation is done in batches.

Arguments

- **x**: Numpy array of test data (if the model has a single input), or list of Numpy arrays (if the model has multiple inputs). If input layers in the model are named, you can also pass a dictionary mapping input names to Numpy arrays. `x` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **y**: Numpy array of target (label) data (if the model has a single output), or list of Numpy arrays (if the model has multiple outputs). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays. `y` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **batch_size**: Integer or `None`. Number of samples per evaluation step. If unspecified, `batch_size` will default to 32.
- **verbose**: 0 or 1. Verbosity mode. 0 = silent, 1 = progress bar.

- **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. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.
- **steps**: Integer or `None`. Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of `None`.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during evaluation. See [callbacks](#).

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.

predict

```
predict(x, batch_size=None, verbose=0, steps=None, callbacks=None)
```

Generates output predictions for the input samples.

Computation is done in batches.

Arguments

- **x**: The input data, as a Numpy array (or list of Numpy arrays if the model has multiple inputs).
- **batch_size**: Integer. If unspecified, it will default to 32.
- **verbose**: Verbosity mode, 0 or 1.
- **steps**: Total number of steps (batches of samples) before declaring the prediction round finished. Ignored with the default value of `None`.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during prediction. See [callbacks](#).

Returns

Numpy array(s) of predictions.

Raises

- **ValueError:** In case of mismatch between the provided input data and the model's expectations, or in case a stateful model receives a number of samples that is not a multiple of the batch size.
-

train_on_batch

```
train_on_batch(x, y, sample_weight=None, class_weight=None)
```

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

Arguments

- **x:** Numpy array of training data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y:** Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **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. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.
- **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.

Returns

Scalar training 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.

test_on_batch

```
test_on_batch(x, y, sample_weight=None)
```

Test the model on a single batch of samples.

Arguments

- **x**: Numpy array of test data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **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. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.

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.

predict_on_batch

```
predict_on_batch(x)
```

Returns predictions for a single batch of samples.

Arguments

- **x**: Input samples, as a Numpy array.

Returns

Numpy array(s) of predictions.

fit_generator

```
fit_generator(generator, steps_per_epoch=None, epochs=1, verbose=1, callbacks=None, validation_data=None)
```

Trains the model on data generated batch-by-batch by a Python generator (or an instance of `Sequence`).

The generator is run in parallel to the model, for efficiency. For instance, this allows you to do real-time data augmentation on images on CPU in parallel to training your model on GPU.

The use of `keras.utils.Sequence` guarantees the ordering and guarantees the single use of every input per epoch when using `use_multiprocessing=True` .

Arguments

- **generator:** A generator or an instance of `Sequence` (`keras.utils.Sequence`) object in order to avoid duplicate data when using multiprocessing. The output of the generator must be either
 - a tuple `(inputs, targets)`
 - a tuple `(inputs, targets, sample_weights)` .

This tuple (a single output of the generator) makes a single batch. Therefore, all arrays in this tuple must have the same length (equal to the size of this batch). Different batches may have different sizes. For example, the last batch of the epoch is commonly smaller than the others, if the size of the dataset is not divisible by the batch size. The generator is expected to loop over its data indefinitely. An epoch finishes when `steps_per_epoch` batches have been seen by the model.

- **steps_per_epoch:** Integer. Total number of steps (batches of samples) to yield from `generator` before declaring one epoch finished and starting the next epoch. It should typically be equal to `ceil(num_samples / batch_size)` . Optional for `Sequence` : if unspecified, will use the `len(generator)` as a number of steps.
- **epochs:** Integer. Number of epochs to train the model. An epoch is an iteration over the entire data provided, as defined by `steps_per_epoch` . 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:** Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.
- **callbacks:** List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See [callbacks](#).
- **validation_data:** This can be either
 - a generator or a `Sequence` object for the validation data
 - tuple `(x_val, y_val)`
 - tuple `(x_val, y_val, val_sample_weights)`

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.

- **validation_steps**: Only relevant if `validation_data` is a generator. Total number of steps (batches of samples) to yield from `validation_data` generator before stopping at the end of every epoch. It should typically be equal to the number of samples of your validation dataset divided by the batch size. Optional for `Sequence`: if unspecified, will use the `len(validation_data)` as a number of steps.
- **validation_freq**: Only relevant if validation data is provided. Integer or `collections.Container` instance (e.g. list, tuple, etc.). If an integer, specifies how many training epochs to run before a new validation run is performed, e.g. `validation_freq=2` runs validation every 2 epochs. If a Container, specifies the epochs on which to run validation, e.g. `validation_freq=[1, 2, 10]` runs validation at the end of the 1st, 2nd, and 10th epochs.
- **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.
- **max_queue_size**: Integer. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers**: Integer. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: Boolean. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.
- **shuffle**: Boolean. Whether to shuffle the order of the batches at the beginning of each epoch. Only used with instances of `Sequence` (`keras.utils.Sequence`). Has no effect when `steps_per_epoch` is not `None`.
- **initial_epoch**: Integer. Epoch at which to start training (useful for resuming a previous training run).

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).

Raises

- **ValueError**: In case the generator yields data in an invalid format.

Example

```
def generate_arrays_from_file(path):
    while True:
        with open(path) as f:
            for line in f:
                # create numpy arrays of input data
                # and labels, from each line in the file
                x1, x2, y = process_line(line)
                yield ({'input_1': x1, 'input_2': x2}, {'output': y})

model.fit_generator(generate_arrays_from_file('/my_file.txt'),
                    steps_per_epoch=10000, epochs=10)
```

evaluate_generator

```
evaluate_generator(generator, steps=None, callbacks=None, max_queue_size=10, workers=1, use_multiproc
```

Evaluates the model on a data generator.

The generator should return the same kind of data as accepted by `test_on_batch`.

Arguments

- **generator**: Generator yielding tuples (inputs, targets) or (inputs, targets, sample_weights) or an instance of Sequence (keras.utils.Sequence) object in order to avoid duplicate data when using multiprocessing.
- **steps**: Total number of steps (batches of samples) to yield from `generator` before stopping. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See [callbacks](#).
- **max_queue_size**: maximum size for the generator queue
- **workers**: Integer. Maximum number of processes to spin up when using process based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: if True, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non picklable arguments to the generator as they can't be passed easily to children processes.
- **verbose**: verbosity mode, 0 or 1.

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.

Raises

- **ValueError:** In case the generator yields data in an invalid format.

predict_generator

```
predict_generator(generator, steps=None, callbacks=None, max_queue_size=10, workers=1, use_multiprocessing=False)
```

Generates predictions for the input samples from a data generator.

The generator should return the same kind of data as accepted by `predict_on_batch`.

Arguments

- **generator:** Generator yielding batches of input samples or an instance of `Sequence` (`keras.utils.Sequence`) object in order to avoid duplicate data when using multiprocessing.
- **steps:** Total number of steps (batches of samples) to yield from `generator` before stopping. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **callbacks:** List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See [callbacks](#).
- **max_queue_size:** Maximum size for the generator queue.
- **workers:** Integer. Maximum number of processes to spin up when using process based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing:** If `True`, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non picklable arguments to the generator as they can't be passed easily to children processes.
- **verbose:** verbosity mode, 0 or 1.

Returns

Numpy array(s) of predictions.

Raises

- **ValueError:** In case the generator yields data in an invalid format.
-

get_layer

```
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.

Raises

- **ValueError:** In case of invalid layer name or index.