tf.keras.models.Model

Class **`Model`**

Defined in `tensorflow/python/keras/_impl/keras/engine/training.py`.

The `Model` class adds training & evaluation routines to a `Container`.

Properties

`activity_regularizer`

Optional regularizer function for the output of this layer.

`dtype`

`graph`

`input`

Retrieves the input tensor(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer.

Returns:

Input tensor or list of input tensors.

Raises:

- `AttributeError`: if the layer is connected to more than one incoming layers.

Raises:

- `RuntimeError`: If called in Eager mode.
- `AttributeError`: If no inbound nodes are found.

`input_mask`

Retrieves the input mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns:

Input mask tensor (potentially None) or list of input mask tensors.

Raises:

- `AttributeError`: if the layer is connected to more than one incoming layers.

`input_shape`

Retrieves the input shape(s) of a layer.

Only applicable if the layer has exactly one input, i.e. if it is connected to one incoming layer, or if all inputs have the same shape.

Returns:

Input shape, as an integer shape tuple (or list of shape tuples, one tuple per input tensor).

Raises:

- `AttributeError`: if the layer has no defined input_shape.
- `RuntimeError`: if called in Eager mode.

`input_spec`

Gets the network's input specs.

Returns:

A list of `InputSpec` instances (one per input to the model) or a single instance if the model has only one input.

`losses`

Retrieve the network's losses.

Will only include losses that are either unconditional, or conditional on inputs to this model (e.g. will not include losses that depend on tensors that aren't inputs to this model).

Returns:

A list of loss tensors.

`name`

`non_trainable_variables`

`non_trainable_weights`

`output`

Retrieves the output tensor(s) of a layer.

Only applicable if the layer has exactly one output, i.e. if it is connected to one incoming layer.

Returns:

Output tensor or list of output tensors.

Raises:

- `AttributeError` : if the layer is connected to more than one incoming layers.
- `RuntimeError` : if called in Eager mode.

`output_mask`

Retrieves the output mask tensor(s) of a layer.

Only applicable if the layer has exactly one inbound node, i.e. if it is connected to one incoming layer.

Returns:

Output mask tensor (potentially None) or list of output mask tensors.

Raises:

- `AttributeError` : if the layer is connected to more than one incoming layers.

`output_shape`

Retrieves the output shape(s) of a layer.

Only applicable if the layer has one output, or if all outputs have the same shape.

Returns:

Output shape, as an integer shape tuple (or list of shape tuples, one tuple per output tensor).

Raises:

- `AttributeError` : if the layer has no defined output shape.
- `RuntimeError` : if called in Eager mode.

`scope_name`

`state_updates`

Returns the `updates` from all layers that are stateful.

This is useful for separating training updates and state updates, e.g. when we need to update a layer's internal state during prediction.

Returns:

A list of update ops.

`stateful`

`trainable_variables`

`trainable_weights`

`updates`

Retrieve the network's updates.

Will only include updates that are either unconditional, or conditional on inputs to this model (e.g. will not include updates that depend on tensors that aren't inputs to this model).

Returns:

A list of update ops.

`uses_learning_phase`

`variables`

Returns the list of all layer variables/weights.

Returns:

A list of variables.

`weights`

Returns the list of all layer variables/weights.

Returns:

A list of variables.

Methods

`__init__`

```
__init__(
    inputs,
    outputs,
    name=None
)
```

`__call__`

```
__call__(
    inputs,
    **kwargs
)
```

Wrapper around self.call(), for handling internal references.

If a Keras tensor is passed: - We call self._add_inbound_node(). - If necessary, we `build` the layer to match the shape of the input(s). - We update the _keras_history of the output tensor(s) with the current layer. This is done as part of _add_inbound_node().

Arguments:

- `inputs`: Can be a tensor or list/tuple of tensors.
- `**kwargs`: Additional keyword arguments to be passed to `call()`.

Returns:

Output of the layer's `call` method.

Raises:

- `ValueError`: in case the layer is missing shape information for its `build` call.

`__deepcopy__`

```
__deepcopy__(memo)
```

`add_loss`

```
add_loss(
    losses,
    inputs=None
)
```

Add loss tensor(s), potentially dependent on layer inputs.

Some losses (for instance, activity regularization losses) may be dependent on the inputs passed when calling a layer. Hence, when reusing a same layer on different inputs `a` and `b`, some entries in `layer.losses` may be dependent on `a` and some on `b`. This method automatically keeps track of dependencies.

The `get_losses_for` method allows to retrieve the losses relevant to a specific set of inputs.

Arguments:

- `losses`: Loss tensor, or list/tuple of tensors.
- `inputs`: Optional input tensor(s) that the loss(es) depend on. Must match the **`inputs`** argument passed to the **`__call__`** method at the time the losses are created. If **`None`** is passed, the losses are assumed to be unconditional, and will apply across all dataflows of the layer (e.g. weight regularization losses).

Raises:

- `RuntimeError`: If called in Eager mode.

add_update

```
add_update(
    updates,
    inputs=None
)
```

Add update op(s), potentially dependent on layer inputs.

Weight updates (for instance, the updates of the moving mean and variance in a BatchNormalization layer) may be dependent on the inputs passed when calling a layer. Hence, when reusing a same layer on different inputs **`a`** and **`b`**, some entries in **`layer.updates`** may be dependent on **`a`** and some on **`b`**. This method automatically keeps track of dependencies.

The **`get_updates_for`** method allows to retrieve the updates relevant to a specific set of inputs.

This call is ignored in Eager mode.

Arguments:

- `updates`: Update op, or list/tuple of update ops.
- `inputs`: Optional input tensor(s) that the update(s) depend on. Must match the **`inputs`** argument passed to the **`__call__`** method at the time the updates are created. If **`None`** is passed, the updates are assumed to be unconditional, and will apply across all dataflows of the layer.

add_variable

```
add_variable(
    name,
    shape,
    dtype=None,
    initializer=None,
    regularizer=None,
    trainable=True,
    constraint=None
)
```

Adds a new variable to the layer, or gets an existing one; returns it.

Arguments:

- `name`: variable name.
- `shape`: variable shape.
- `dtype`: The type of the variable. Defaults to **`self.dtype`** or **`float32`**.

- `initializer` : initializer instance (callable).
- `regularizer` : regularizer instance (callable).
- `trainable` : whether the variable should be part of the layer's "trainable_variables" (e.g. variables, biases) or "non_trainable_variables" (e.g. BatchNorm mean, stddev).
- `constraint` : constraint instance (callable).

Returns:

The created variable.

Raises:

- `RuntimeError` : If called in Eager mode with regularizers.

`add_weight`

```
add_weight(
    name,
    shape,
    dtype=None,
    initializer=None,
    regularizer=None,
    trainable=True,
    constraint=None
)
```

Adds a weight variable to the layer.

Arguments:

- `name` : String, the name for the weight variable.
- `shape` : The shape tuple of the weight.
- `dtype` : The dtype of the weight.
- `initializer` : An Initializer instance (callable).
- `regularizer` : An optional Regularizer instance.
- `trainable` : A boolean, whether the weight should be trained via backprop or not (assuming that the layer itself is also trainable).
- `constraint` : An optional Constraint instance.

Returns:

The created weight variable.

`apply`

```
apply(
    inputs,
    *args,
    **kwargs
)
```

Apply the layer on a input.

This simply wraps `self.__call__`.

Arguments:

- `inputs`: Input tensor(s).
- `*args`: additional positional arguments to be passed to `self.call`.
- `**kwargs`: additional keyword arguments to be passed to `self.call`.

Returns:

Output tensor(s).

`build`

```
build(_)
```

Creates the variables of the layer.

`call`

```
call(
    inputs,
    mask=None
)
```

Call the model on new inputs.

In this case `call` just reapplies all ops in the graph to the new inputs (e.g. build a new computational graph from the provided inputs).

Arguments:

- `inputs`: A tensor or list of tensors.
- `mask`: A mask or list of masks. A mask can be either a tensor or None (no mask).

Returns:

A tensor if there is a single output, or a list of tensors if there are more than one outputs.

`compile`

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

Configures the model for training.

Arguments:

- `optimizer`: String (name of optimizer) or optimizer object. 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`.

`compute_mask`

```
compute_mask(
    inputs,
    mask
)
```

`count_params`

```
count_params()
```

Count the total number of scalars composing the weights.

Returns:

An integer count.

Raises:

- `ValueError`: if the layer isn't yet built (in which case its weights aren't yet defined).

`evaluate`

```
evaluate(
    x,
    y,
    batch_size=None,
    verbose=1,
    sample_weight=None,
    steps=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, 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.
- `batch_size`: Integer. If unspecified, it will default to 32.
- `verbose`: Verbosity mode, 0 or 1.
- `sample_weight`: Array of weights to weight the contribution of different samples to the loss and metrics.
- `steps`: Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of `None`.

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 of invalid argument values.

`evaluate_generator`

```
evaluate_generator(
    generator,
    steps,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
    **kwargs
)
```

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.
- `max_queue_size`: maximum size for the generator queue
- `workers`: maximum number of processes to spin up when using process based threading
- `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.
- `**kwargs`: support for legacy arguments.

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.

`fit`

```
fit(
    x=None,
    y=None,
    batch_size=None,
    epochs=1,
    verbose=1,
    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
)
```

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

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.
- `batch_size`: Integer or **`None`**. Number of samples per gradient update. If unspecified, it will default to 32.
- `epochs`: Integer, the number of times to iterate over the training data arrays.
- `verbose`: 0, 1, or 2. Verbosity mode. 0 = silent, 1 = verbose, 2 = one log line per epoch.
- `callbacks`: List of callbacks to be called during training. 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.
- `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. This could be a tuple (x_val, y_val) or a tuple (x_val, y_val, val_sample_weights).
- `shuffle`: Boolean, whether to shuffle the training data before each epoch. Has no effect when **`steps_per_epoch`** is not **`None`**.
- `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.
- `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().
- `initial_epoch`: Epoch at which to start training (useful for resuming a previous training run)
- `steps_per_epoch`: 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 unique 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.

Returns:

A **`History`** instance. Its **`history`** attribute contains all information collected during training.

Raises:

- `ValueError`: In case of mismatch between the provided input data and what the model expects.

`fit_generator`

```
fit_generator(
    generator,
    steps_per_epoch,
    epochs=1,
    verbose=1,
    callbacks=None,
    validation_data=None,
    validation_steps=None,
    class_weight=None,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
    shuffle=True,
    initial_epoch=0,
    **kwargs
)
```

Fits the model on data yielded batch-by-batch by a Python generator.

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). All arrays should contain the same number of samples. 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`: 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 the number of unique samples if your dataset divided by the batch size.
- `epochs`: Integer, total number of iterations on the data.
- `verbose`: Verbosity mode, 0, 1, or 2.
- `callbacks`: List of callbacks to be called during training.
- `validation_data`: This can be either - a generator for the validation data - a tuple (inputs, targets) - a tuple (inputs, targets, sample_weights).
- `validation_steps`: Only relevant if `validation_data` is a generator. Total number of steps (batches of samples) to yield from `generator` before stopping.
- `class_weight`: Dictionary mapping class indices to a weight for the class.
- `max_queue_size`: Maximum size for the generator queue
- `workers`: Maximum number of processes to spin up when using process based threading
- `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.
- `shuffle`: Whether to shuffle the data at the beginning of each epoch. Only used with instances of `Sequence` (keras.utils.Sequence).
- `initial_epoch`: Epoch at which to start training (useful for resuming a previous training run)
- `**kwargs`: support for legacy arguments.

Returns:

```
A `History` object.
```

Example:

```
    def generate_arrays_from_file(path):
        while 1:
            f = open(path)
            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})
            f.close()

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

Raises:

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

`from_config`

```
from_config(
    cls,
    config,
    custom_objects=None
)
```

Instantiates a Model from its config (output of `get_config()`).

Arguments:

- `config` : Model config dictionary.
- `custom_objects` : Optional dictionary mapping names (strings) to custom classes or functions to be considered during deserialization.

Returns:

A model instance.

Raises:

- `ValueError` : In case of improperly formatted config dict.

`get_config`

```
get_config()
```

`get_input_at`

```
get_input_at(node_index)
```

Retrieves the input tensor(s) of a layer at a given node.

Arguments:

- `node_index` : Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns:

A tensor (or list of tensors if the layer has multiple inputs).

Raises:

- `RuntimeError` : If called in Eager mode.

`get_input_mask_at`

```
get_input_mask_at(node_index)
```

Retrieves the input mask tensor(s) of a layer at a given node.

Arguments:

- `node_index`: Integer, index of the node from which to retrieve the attribute. E.g. **`node_index=0`** will correspond to the first time the layer was called.

Returns:

A mask tensor (or list of tensors if the layer has multiple inputs).

`get_input_shape_at`

```
get_input_shape_at(node_index)
```

Retrieves the input shape(s) of a layer at a given node.

Arguments:

- `node_index`: Integer, index of the node from which to retrieve the attribute. E.g. **`node_index=0`** will correspond to the first time the layer was called.

Returns:

A shape tuple (or list of shape tuples if the layer has multiple inputs).

Raises:

- `RuntimeError`: If called in Eager mode.

`get_layer`

```
get_layer(
    name=None,
    index=None
)
```

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

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.

`get_losses_for`

```
get_losses_for(inputs)
```

Retrieves losses relevant to a specific set of inputs.

Arguments:

- `inputs`: Input tensor or list/tuple of input tensors. Must match the **`inputs`** argument passed to the **`__call__`** method at the time the losses were created. If you pass **`inputs=None`**, unconditional losses are returned, such as weight regularization losses.

Returns:

List of loss tensors of the layer that depend on **`inputs`**.

Raises:

- `RuntimeError`: If called in Eager mode.

`get_output_at`

```
get_output_at(node_index)
```

Retrieves the output tensor(s) of a layer at a given node.

Arguments:

- `node_index`: Integer, index of the node from which to retrieve the attribute. E.g. **`node_index=0`** will correspond to the first time the layer was called.

Returns:

A tensor (or list of tensors if the layer has multiple outputs).

Raises:

- `RuntimeError`: If called in Eager mode.

`get_output_mask_at`

```
get_output_mask_at(node_index)
```

Retrieves the output mask tensor(s) of a layer at a given node.

Arguments:

- `node_index`: Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns:

A mask tensor (or list of tensors if the layer has multiple outputs).

`get_output_shape_at`

```
get_output_shape_at(node_index)
```

Retrieves the output shape(s) of a layer at a given node.

Arguments:

- `node_index`: Integer, index of the node from which to retrieve the attribute. E.g. `node_index=0` will correspond to the first time the layer was called.

Returns:

A shape tuple (or list of shape tuples if the layer has multiple outputs).

Raises:

- `RuntimeError`: If called in Eager mode.

`get_updates_for`

```
get_updates_for(inputs)
```

Retrieves updates relevant to a specific set of inputs.

Arguments:

- `inputs`: Input tensor or list/tuple of input tensors. Must match the `inputs` argument passed to the `__call__` method at the time the updates were created. If you pass `inputs=None`, unconditional updates are returned.

Returns:

List of update ops of the layer that depend on `inputs`.

Raises:

- `RuntimeError`: If called in Eager mode.

`get_weights`

```
get_weights()
```

Retrieves the weights of the model.

Returns:

A flat list of Numpy arrays.

`load_weights`

```
load_weights(
    filepath,
    by_name=False
)
```

Loads all layer weights from a HDF5 save file.

If `by_name` is False (default) weights are loaded based on the network's topology, meaning 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.

If `by_name` is True, weights are loaded into layers only if they share the same name. This is useful for fine-tuning or transfer-learning models where some of the layers have changed.

Arguments:

- `filepath`: String, path to the weights file to load.
- `by_name`: Boolean, whether to load weights by name or by topological order.

Raises:

- `ImportError`: If h5py is not available.

`predict`

```
predict(
    x,
    batch_size=None,
    verbose=0,
    steps=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 outputs).
- `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`.

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.

`predict_generator`

```
predict_generator(
    generator,
    steps,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
    verbose=0,
    **kwargs
)
```

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.
- `max_queue_size`: Maximum size for the generator queue.
- `workers`: Maximum number of processes to spin up when using process based threading
- `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.
- `**kwargs`: support for legacy arguments.

Returns:

Numpy array(s) of predictions.

Raises:

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

`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.

`reset_states`

```
reset_states()
```

`save`

```
save(
    filepath,
    overwrite=True,
    include_optimizer=True
)
```

Save the model to a single HDF5 file.

The savefile includes: - The model architecture, allowing to re-instantiate the model. - The model weights. - The state of the optimizer, allowing to resume training exactly where you left off.

This allows you to save the entirety of the state of a model in a single file.

Saved models can be reinstantiated via **`keras.models.load_model`**. The model returned by **`load_model`** is a compiled model ready to be used (unless the saved model was never compiled in the first place).

Arguments:

- `filepath`: String, path to the file to save the weights to.
- `overwrite`: Whether to silently overwrite any existing file at the target location, or provide the user with a manual prompt.
- `include_optimizer`: If True, save optimizer's state together.

Example:

```
from keras.models import load_model

model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
del model  # deletes the existing model

# returns a compiled model
# identical to the previous one
model = load_model('my_model.h5')
```

`save_weights`

```
save_weights(
    filepath,
    overwrite=True
)
```

Dumps all layer weights to a HDF5 file.

The weight file has: - **`layer_names`** (attribute), a list of strings (ordered names of model layers). - For every layer, a **`group`** named **`layer.name`** - For every such layer group, a group attribute **`weight_names`**, a list of strings (ordered names of weights tensor of the layer). - For every weight in the layer, a dataset storing the weight value, named after the weight tensor.

Arguments:

- `filepath`: String, path to the file to save the weights to.
- `overwrite`: Whether to silently overwrite any existing file at the target location, or provide the user with a manual prompt.

Raises:

- `ImportError`: If h5py is not available.

`set_weights`

```
set_weights(weights)
```

Sets the weights of the model.

Arguments:

- `weights`: A list of Numpy arrays with shapes and types matching the output of **`model.get_weights()`**.

`summary`

```
summary(
    line_length=None,
    positions=None,
    print_fn=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, defaults to **`[.33, .55, .67, 1.]`**.
- `print_fn`: Print function to use. Defaults 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.

`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.

to_json

```
to_json(**kwargs)
```

Returns a JSON string containing the network configuration.

To load a network from a JSON save file, use **`keras.models.model_from_json(json_string, custom_objects={})`**.

Arguments:

- `**kwargs`: Additional keyword arguments to be passed to **`json.dumps()`**.

Returns:

A JSON string.

to_yaml

```
to_yaml(**kwargs)
```

Returns a yaml string containing the network configuration.

To load a network from a yaml save file, use **`keras.models.model_from_yaml(yaml_string, custom_objects={})`**.

`custom_objects` should be a dictionary mapping the names of custom losses / layers / etc to the corresponding functions / classes.

Arguments:

- `**kwargs`: Additional keyword arguments to be passed to **`yaml.dump()`**.

Returns:

A YAML string.

Raises:

- `ImportError`: if yaml module is not found.

`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.