tf.keras.models.Sequential

Contents

Class **`Sequential`**

Inherits From: `Model`

Defined in `tensorflow/python/keras/_impl/keras/models.py`.

Linear stack of layers.

Arguments:

- `layers`: list of layers to add to the model.

Note

```
The first layer passed to a Sequential model
should have a defined input shape. What that
means is that it should have received an `input_shape`
or `batch_input_shape` argument,
or for some type of layers (recurrent, Dense...)
an `input_dim` argument.
```

Example:

~~~
```python
    model = Sequential()
    # first layer must have a defined input shape
    model.add(Dense(32, input_dim=500))
    # afterwards, Keras does automatic shape inference
    model.add(Dense(32))

    # also possible (equivalent to the above):
    model = Sequential()
    model.add(Dense(32, input_shape=(500,)))
    model.add(Dense(32))

    # also possible (equivalent to the above):
    model = Sequential()
    # here the batch dimension is None,
    # which means any batch size will be accepted by the model.
    model.add(Dense(32, batch_input_shape=(None, 500)))
    model.add(Dense(32))
```
~~~

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`

`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.

`regularizers`

`scope_name`

`state_updates`

`stateful`

`trainable`

`trainable_variables`

`trainable_weights`

`updates`

`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__(
    layers=None,
    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

```
add(layer)
```

Adds a layer instance on top of the layer stack.

Arguments:

- `layer` : layer instance.

Raises:

- `TypeError` : If `layer` is not a layer instance.
- `ValueError` : In case the `layer` argument does not know its input shape.

- `ValueError`: In case the `layer` argument has multiple output tensors, or is already connected somewhere else (forbidden in `Sequential` models).

`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(input_shape=None)
```

`call`

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

`compile`

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

Configures the learning process.

Arguments:

- `optimizer`: str (name of optimizer) or optimizer object. See optimizers.
- `loss`: str (name of objective function) or objective function. See losses.
- `metrics`: list of metrics to be evaluated by the model during training and testing. Typically you will use `metrics=['accuracy']`. See metrics.
- `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).
- `weighted_metrics`: list of metrics to be evaluated and weighted by `sample_weight` or `class_weight` during training and testing.
- `**kwargs`: These are passed into `tf.Session.run`.

Example: `python model = Sequential() model.add(Dense(32, input_shape=(500,))) model.add(Dense(10, activation='softmax')) model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])`

`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=32,
    verbose=1,
    sample_weight=None
)
```

Computes the loss on some input data, batch by batch.

Arguments:

- `x` : input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- `y` : labels, as a Numpy array.
- `batch_size` : integer. Number of samples per gradient update.
- `verbose` : verbosity mode, 0 or 1.
- `sample_weight` : sample weights, as a Numpy array.

Returns:

Scalar test loss (if the model has no metrics) or list of scalars (if the model computes other metrics). The attribute **`model.metrics_names`** will give you the display labels for the scalar outputs.

Raises:

- `RuntimeError` : if the model was never compiled.

`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)
- `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
- `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 no metrics) or list of scalars (if the model computes other metrics). The attribute **`model.metrics_names`** will give you the display labels for the scalar outputs.

Raises:

- `RuntimeError` : if the model was never compiled.
- `ValueError` : In case the generator yields data in an invalid format.

fit

```
fit(
    x,
    y,
    batch_size=32,
    epochs=10,
    verbose=1,
    callbacks=None,
    validation_split=0.0,
    validation_data=None,
    shuffle=True,
    class_weight=None,
    sample_weight=None,
    initial_epoch=0
)
```

Trains the model for a fixed number of epochs.

Arguments:

- `x` : input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- `y` : labels, as a Numpy array.
- `batch_size` : integer. Number of samples per gradient update.
- `epochs` : integer, the number of epochs to train the model.
- `verbose` : 0 for no logging to stdout, 1 for progress bar logging, 2 for one log line per epoch.
- `callbacks` : list of **`keras.callbacks.Callback`** instances. List of callbacks to apply during training. See callbacks.
- `validation_split` : float (0. < x < 1). Fraction of the data to use as held-out validation data.
- `validation_data` : tuple (x_val, y_val) or tuple (x_val, y_val, val_sample_weights) to be used as held-out validation data. Will override validation_split.
- `shuffle` : boolean or str (for 'batch'). Whether to shuffle the samples at each epoch. 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks.
- `class_weight` : dictionary mapping classes to a weight value, used for scaling the loss function (during training only).
- `sample_weight` : Numpy array of weights for the training samples, used for scaling 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` : 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:

- `RuntimeError` : if the model was never compiled.

`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,
    initial_epoch=0,
    **kwargs
)
```

Fits the model on data generated 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.

Arguments:

- `generator` : A generator. 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 of 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. Number of steps to yield from validation generator at the end of every epoch. It should typically be equal to the number of unique samples of your validation dataset divided by the batch size.
- `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
- `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.

- `initial_epoch`: Epoch at which to start training (useful for resuming a previous training run)
- `**kwargs`: support for legacy arguments.

Returns:

A `History` object.

Raises:

- `RuntimeError`: if the model was never compiled.
- `ValueError`: In case the generator yields data in an invalid format.

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
                x, y = process_line(line)
                yield (x, y)
                f.close()

    model.fit_generator(generate_arrays_from_file('/my_file.txt'),
                        steps_per_epoch=1000, epochs=10)
```

`from_config`

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

`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
)
```

Retrieve a layer that is part of the model.

Returns a layer based on either its name (unique) or its index in the graph. 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.

`get_losses_for`

```
get_losses_for(inputs)
```

`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)
```

`get_weights`

```
get_weights()
```

Retrieves the weights of the model.

Returns:

A flat list of Numpy arrays (one array per model weight).

`load_weights`

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

`pop`

```
pop()
```

Removes the last layer in the model.

Raises:

- `TypeError`: if there are no layers in the model.

`predict`

```
predict(
    x,
    batch_size=32,
    verbose=0
)
```

Generates output predictions for the input samples.

The input samples are processed batch by batch.

Arguments:

- `x`: the input data, as a Numpy array.
- `batch_size`: integer.
- `verbose`: verbosity mode, 0 or 1.

Returns:

A Numpy array of predictions.

`predict_classes`

```
predict_classes(
    x,
    batch_size=32,
    verbose=1
)
```

Generate class predictions for the input samples.

The input samples are processed batch by batch.

Arguments:

- `x`: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- `batch_size`: integer.
- `verbose`: verbosity mode, 0 or 1.

Returns:

A numpy array of class predictions.

`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.
- `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
- `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:

A Numpy array 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 data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).

Returns:

A Numpy array of predictions.

`predict_proba`

```
predict_proba(
    x,
    batch_size=32,
    verbose=1
)
```

Generates class probability predictions for the input samples.

The input samples are processed batch by batch.

Arguments:

- `x` : input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- `batch_size` : integer.
- `verbose` : verbosity mode, 0 or 1.

Returns:

A Numpy array of probability 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
)
```

`set_weights`

```
set_weights(weights)
```

Sets the weights of the model.

Arguments:

- `weights` : Should be 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
)
```

Evaluates the model over a single batch of samples.

Arguments:

- `x` : input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- `y` : labels, as a Numpy array.
- `sample_weight` : sample weights, as a Numpy array.

Returns:

Scalar test loss (if the model has no metrics) or list of scalars (if the model computes other metrics). The attribute **`model.metrics_names`** will give you the display labels for the scalar outputs.

Raises:

- `RuntimeError` : if the model was never compiled.

`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,
    class_weight=None,
    sample_weight=None
)
```

Single gradient update over one batch of samples.

Arguments:

- `x` : input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- `y` : labels, as a Numpy array.
- `class_weight` : dictionary mapping classes to a weight value, used for scaling the loss function (during training only).
- `sample_weight` : sample weights, as a Numpy array.

Returns:

Scalar training loss (if the model has no metrics) or list of scalars (if the model computes other metrics). The attribute **`model.metrics_names`** will give you the display labels for the scalar outputs.

Raises:

- `RuntimeError` : if the model was never compiled.