Model training APIs

`compile` method [source]

```
Model.compile(
    optimizer="rmsprop",
    loss=None,
    metrics=None,
    loss_weights=None,
    weighted_metrics=None,
    run_eagerly=None,
    steps_per_execution=None,
    jit_compile=None,
    **kwargs
)
```

Configures the model for training.

Example

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=[tf.keras.metrics.BinaryAccuracy(),
                       tf.keras.metrics.FalseNegatives()])
```

Arguments

- **optimizer**: String (name of optimizer) or optimizer instance. See `tf.keras.optimizers`.
- **loss**: Loss function. May be a string (name of loss function), or a `tf.keras.losses.Loss` instance. See `tf.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. If a custom `Loss` instance is used and reduction is set to None, return value has shape `(batch_size, d0, .. dN-1)` i.e. per-sample or per-timestep loss values; otherwise, it is a scalar. 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, unless `loss_weights` is specified.
- **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 `tf.keras.metrics.Metric` instance. See `tf.keras.metrics`. Typically you will use `metrics=['accuracy']`. A function is any callable with the signature `result = fn(y_true, y_pred)`. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as `metrics={'output_a': 'accuracy', 'output_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 `tf.keras.metrics.BinaryAccuracy`,

`tf.keras.metrics.CategoricalAccuracy`, `tf.keras.metrics.SparseCategoricalAccuracy` based on the loss function used and the model output shape. We do a similar conversion for the strings 'crossentropy' and 'ce' as well.

- **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.
- **weighted_metrics**: List of metrics to be evaluated and weighted by `sample_weight` or `class_weight` during training and testing.
- **run_eagerly**: Bool. Defaults to `False`. If `True`, this `Model`'s logic will not be wrapped in a `tf.function`. Recommended to leave this as `None` unless your `Model` cannot be run inside a `tf.function`. `run_eagerly=True` is not supported when using `tf.distribute.experimental.ParameterServerStrategy`.
- **steps_per_execution**: Int. Defaults to 1. The number of batches to run during each `tf.function` call. Running multiple batches inside a single `tf.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 `tf.function` execution).
- **jit_compile**: If `True`, compile the model training step with XLA. XLA is an optimizing compiler for machine learning. `jit_compile` is not enabled for by default. This option cannot be enabled with `run_eagerly=True`. Note that `jit_compile=True` is may not necessarily work for all models. For more information on supported operations please refer to the XLA documentation. Also refer to known XLA issues for more details.
- ****kwargs**: Arguments supported for backwards compatibility only.

`fit` method [source]

```
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,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
)
```

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

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 TensorFlow 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. Should return a tuple of either `(inputs, targets)` or `(inputs, targets, sample_weights)`.
  - A generator or `keras.utils.Sequence` returning `(inputs, targets)` or `(inputs, targets, sample_weights)`.
  - A `tf.keras.utils.experimental.DatasetCreator`, which wraps a callable that takes a single argument of type `tf.distribute.InputContext`, and returns a `tf.data.Dataset`. `DatasetCreator` should be used when users prefer to specify the per-replica batching and sharding logic for the `Dataset`. See `tf.keras.utils.experimental.DatasetCreator` doc for more information. A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given below. If using `tf.distribute.experimental.ParameterServerStrategy`, only `DatasetCreator` type is supported for `x`.
- **y**: Target data. Like the input data `x`, it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with `x` (you cannot have Numpy inputs and tensor targets, or inversely). If `x` is a dataset, generator, or `keras.utils.Sequence` 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.Sequence` 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' defaults to 1 for most cases, but 2 when used with `ParameterServerStrategy`. Note that the progress bar is not particularly useful when logged to a file, so verbose=2 is recommended when not running interactively (eg, in a production environment).
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See `tf.keras.callbacks`. Note `tf.keras.callbacks.ProgbarLogger` and `tf.keras.callbacks.History` callbacks are created automatically and need not be passed into `model.fit`. `tf.keras.callbacks.ProgbarLogger` is created or not based on `verbose` argument to `model.fit`. Callbacks with batch-level calls are currently unsupported with `tf.distribute.experimental.ParameterServerStrategy`, and users are advised to implement epoch-level calls instead with an appropriate `steps_per_epoch` value.
- **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.Sequence` instance. If both `validation_data` and

`validation_split` are provided, `validation_data` will override `validation_split`. `validation_split` is not yet supported with `tf.distribute.experimental.ParameterServerStrategy`.

- **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`. `validation_data` 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`. - A Python generator or `keras.utils.Sequence` returning `(inputs, targets)` or `(inputs, targets, sample_weights)`. `validation_data` is not yet supported with `tf.distribute.experimental.ParameterServerStrategy`.
- **shuffle**: Boolean (whether to shuffle the training data before each epoch) or str (for 'batch'). This argument is ignored when x is a generator or an object of tf.data.Dataset. '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. This argument is not supported when x is a dataset, generator, or `keras.utils.Sequence` instance, instead provide the sample_weights as the third element of x.
- **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. If x is a `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. This argument is not supported with array inputs. When using `tf.distribute.experimental.ParameterServerStrategy`: * `steps_per_epoch=None` is not supported.
- **validation_steps**: Only relevant if `validation_data` is provided and is a `tf.data` dataset. 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, generators, or `keras.utils.Sequence` instances (since they generate batches).
- **validation_freq**: Only relevant if validation data is provided. Integer or `collections.abc.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.
- **max_queue_size**: Integer. Used for generator or `keras.utils.Sequence` input only. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers**: Integer. Used for generator or `keras.utils.Sequence` input only. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1.
- **use_multiprocessing**: Boolean. Used for generator or `keras.utils.Sequence` input only. 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.

Unpacking behavior for iterator-like inputs: A common pattern is to pass a tf.data.Dataset, generator, or tf.keras.utils.Sequence to the x argument of fit, which will in fact yield not only features (x) but optionally targets (y) and sample weights. 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. As a result the data processing code will simply raise a ValueError if it encounters a namedtuple. (Along with instructions to remedy the issue.)

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**: 1. If the model was never compiled or, 2. If `model.fit` is wrapped in `tf.function`.
- **ValueError**: In case of mismatch between the provided input data and what the model expects or when the input data is empty.

`evaluate` method

[source]

```
Model.evaluate(
    x=None,
    y=None,
    batch_size=None,
    verbose="auto",
    sample_weight=None,
    steps=None,
    callbacks=None,
    max_queue_size=10,
```

```
    workers=1,
    use_multiprocessing=False,
    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 TensorFlow 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. Should return a tuple of either `(inputs, targets)` or `(inputs, targets, sample_weights)`.
  - A generator or `keras.utils.Sequence` returning `(inputs, targets)` or `(inputs, targets, sample_weights)`. A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given in the `Unpacking behavior for iterator-like inputs` section of `Model.fit`.
- **y**: Target data. Like the input data `x`, it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with `x` (you cannot have Numpy inputs and tensor targets, or inversely). If `x` is a dataset, generator or `keras.utils.Sequence` 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.Sequence` instances (since they generate batches).
- **verbose**: `"auto"`, 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = single line. `"auto"` defaults to 1 for most cases, and to 2 when used with `ParameterServerStrategy`. 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).
- **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 and `steps` is None, 'evaluate' will run until the dataset is exhausted. This argument is not supported with array inputs.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during evaluation. See callbacks.
- **max_queue_size**: Integer. Used for generator or `keras.utils.Sequence` input only. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers**: Integer. Used for generator or `keras.utils.Sequence` input only. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1.
- **use_multiprocessing**: Boolean. Used for generator or `keras.utils.Sequence` input

only. 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.
- **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.
- ****kwargs**: Unused at this time.

See the discussion of `Unpacking behavior for iterator-like inputs` for `Model.fit`.

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

- **RuntimeError**: If `model.evaluate` is wrapped in a `tf.function`.

`predict` method [source]

```
Model.predict(
    x,
    batch_size=None,
    verbose="auto",
    steps=None,
    callbacks=None,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
)
```

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 `tf.keras.layers.BatchNormalization` that behave differently during inference. You may pair the individual model call with a `tf.function` for additional performance inside your inner loop. If you need access to numpy array values instead of tensors after your model call, you can use `tensor.numpy()` to get the numpy array value of an eager tensor.

Also, note the fact that test loss is not affected by regularization layers like noise and dropout.

Note: See this FAQ entry 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 TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).
    - A `tf.data` dataset.
    - A generator or `keras.utils.Sequence` instance. A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given in the `Unpacking behavior for iterator-like inputs` section of `Model.fit`.
- **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.Sequence` instances (since they generate batches).
- **verbose**: `"auto"`, 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = single line. `"auto"` defaults to 1 for most cases, and to 2 when used with `ParameterServerStrategy`. 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).
- **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 and `steps` is None, `predict()` will run until the input dataset is exhausted.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during prediction. See callbacks.
- **max_queue_size**: Integer. Used for generator or `keras.utils.Sequence` input only. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers**: Integer. Used for generator or `keras.utils.Sequence` input only. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1.
- **use_multiprocessing**: Boolean. Used for generator or `keras.utils.Sequence` input only. 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.

See the discussion of `Unpacking behavior for iterator-like inputs` for `Model.fit`. Note that Model.predict uses the same interpretation rules as `Model.fit` and `Model.evaluate`, so inputs must be unambiguous for all three methods.

Returns

Numpy array(s) of predictions.

Raises

- **RuntimeError**: If `model.predict` is wrapped in a `tf.function`.
- **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` method

[source]

```
Model.train_on_batch(
    x,
    y=None,
    sample_weight=None,
    class_weight=None,
    reset_metrics=True,
    return_dict=False,
)
```

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

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 TensorFlow 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.
- **y**: Target data. Like the input data `x`, it could be either Numpy array(s) or TensorFlow tensor(s).
- **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.
- **reset_metrics**: If `True`, the metrics returned will be only for this batch. If `False`, the metrics will be statefully accumulated across batches.
- **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 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.

Raises

- **RuntimeError**: If `model.train_on_batch` is wrapped in a `tf.function`.

`test_on_batch` method [source]

```
Model.test_on_batch(
    x, y=None, sample_weight=None, reset_metrics=True, return_dict=False
)
```

Test the model on a single batch of samples.

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 TensorFlow 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.
- **y**: Target data. Like the input data `x`, it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with `x` (you cannot have Numpy inputs and tensor targets, or inversely).
- **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.
- **reset_metrics**: If `True`, the metrics returned will be only for this batch. If `False`, the metrics will be statefully accumulated across batches.
- **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.

Raises

- **RuntimeError**: If `model.test_on_batch` is wrapped in a `tf.function`.

`predict_on_batch` method

[source]

`Model.predict_on_batch(x)`

Returns predictions for a single batch of samples.

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 TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).

Returns

Numpy array(s) of predictions.

Raises

- **RuntimeError**: If `model.predict_on_batch` is wrapped in a `tf.function`.

`run_eagerly` property

`tf.keras.Model.run_eagerly`

Settable attribute indicating whether the model should run eagerly.

Running eagerly means that your model will be run step by step, like Python code. Your model might run slower, but it should become easier for you to debug it by stepping into individual layer calls.

By default, we will attempt to compile your model to a static graph to deliver the best execution performance.

Returns

Boolean, whether the model should run eagerly.
