The base Layer class

`Layer` class

[source]

```
tf.keras.layers.Layer(
    trainable=True, name=None, dtype=None, dynamic=False, **kwargs
)
```

This is the class from which all layers inherit.

A layer is a callable object that takes as input one or more tensors and that outputs one or more tensors. It involves *computation*, defined in the `call()` method, and a *state* (weight variables). State can be created in various places, at the convenience of the subclass implementer:

- in `__init__()`;
- in the optional `build()` method, which is invoked by the first `__call__()` to the layer, and supplies the shape(s) of the input(s), which may not have been known at initialization time;
- in the first invocation of `call()`, with some caveats discussed below.

Users will just instantiate a layer and then treat it as a callable.

Arguments

- **trainable**: Boolean, whether the layer's variables should be trainable.
- **name**: String name of the layer.
- **dtype**: The dtype of the layer's computations and weights. Can also be a `tf.keras.mixed_precision.Policy`, which allows the computation and weight dtype to differ. Default of `None` means to use `tf.keras.mixed_precision.global_policy()`, which is a float32 policy unless set to different value.
- **dynamic**: Set this to `True` if your layer should only be run eagerly, and should not be used to generate a static computation graph. This would be the case for a Tree-RNN or a recursive network, for example, or generally for any layer that manipulates tensors using Python control flow. If `False`, we assume that the layer can safely be used to generate a static computation graph.

Attributes

- **name**: The name of the layer (string).
- **dtype**: The dtype of the layer's weights.
- **variable_dtype**: Alias of `dtype`.
- **compute_dtype**: The dtype of the layer's computations. Layers automatically cast inputs to this dtype which causes the computations and output to also be in this dtype. When mixed precision is used with a `tf.keras.mixed_precision.Policy`, this will be different than `variable_dtype`.
- **dtype_policy**: The layer's dtype policy. See the `tf.keras.mixed_precision.Policy` documentation for details.
- **trainable_weights**: List of variables to be included in backprop.
- **non_trainable_weights**: List of variables that should not be included in backprop.
- **weights**: The concatenation of the lists trainable_weights and non_trainable_weights (in this order).
- **trainable**: Whether the layer should be trained (boolean), i.e. whether its potentially-trainable weights should be returned as part of `layer.trainable_weights`.
- **input_spec**: Optional (list of) `InputSpec` object(s) specifying the constraints on inputs that can be accepted by the layer.

We recommend that descendants of `Layer` implement the following methods:

- `__init__()`: Defines custom layer attributes, and creates layer weights that do not depend on input shapes, using `add_weight()`, or other state.
- `build(self, input_shape)`: This method can be used to create weights that depend on the shape(s) of the input(s), using `add_weight()`, or other state. `__call__()` will automatically build the layer (if it has not been built yet) by calling `build()`.

- `call(self, inputs, *args, **kwargs)`: Called in `__call__` after making sure `build()` has been called. `call()` performs the logic of applying the layer to the `inputs`. The first invocation may additionally create state that could not be conveniently created in `build()`; see its docstring for details. Two reserved keyword arguments you can optionally use in `call()` are: - `training` (boolean, whether the call is in inference mode or training mode). See more details in the layer/model subclassing guide - `mask` (boolean tensor encoding masked timesteps in the input, used in RNN layers). See more details in the layer/model subclassing guide A typical signature for this method is `call(self, inputs)`, and user could optionally add `training` and `mask` if the layer need them. `*args` and `**kwargs` is only useful for future extension when more input parameters are planned to be added.
- `get_config(self)`: Returns a dictionary containing the configuration used to initialize this layer. If the keys differ from the arguments in `__init__`, then override `from_config(self)` as well. This method is used when saving the layer or a model that contains this layer.

Examples

Here's a basic example: a layer with two variables, `w` and `b`, that returns `y = w . x + b`. It shows how to implement `build()` and `call()`. Variables set as attributes of a layer are tracked as weights of the layers (in `layer.weights`).

```
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):  # Create the state of the layer (weights)
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(
        initial_value=w_init(shape=(input_shape[-1], self.units),
                             dtype='float32'),
        trainable=True)
    b_init = tf.zeros_initializer()
    self.b = tf.Variable(
        initial_value=b_init(shape=(self.units,), dtype='float32'),
        trainable=True)

  def call(self, inputs):  # Defines the computation from inputs to outputs
      return tf.matmul(inputs, self.w) + self.b

# Instantiates the layer.
linear_layer = SimpleDense(4)

# This will also call `build(input_shape)` and create the weights.
y = linear_layer(tf.ones((2, 2)))
assert len(linear_layer.weights) == 2

# These weights are trainable, so they're listed in `trainable_weights`:
assert len(linear_layer.trainable_weights) == 2
```

Note that the method `add_weight()` offers a shortcut to create weights:

```
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      self.units = units

  def build(self, input_shape):
      self.w = self.add_weight(shape=(input_shape[-1], self.units),
                               initializer='random_normal',
                               trainable=True)
      self.b = self.add_weight(shape=(self.units,),
                               initializer='random_normal',
                               trainable=True)

  def call(self, inputs):
      return tf.matmul(inputs, self.w) + self.b
```

Besides trainable weights, updated via backpropagation during training, layers can also have non-trainable weights. These weights are meant to be updated manually during `call()`. Here's a example layer that computes the running sum of its inputs:

```
class ComputeSum(Layer):

  def __init__(self, input_dim):
      super(ComputeSum, self).__init__()
      # Create a non-trainable weight.
      self.total = tf.Variable(initial_value=tf.zeros((input_dim,)),
                               trainable=False)

  def call(self, inputs):
      self.total.assign_add(tf.reduce_sum(inputs, axis=0))
      return self.total

my_sum = ComputeSum(2)
x = tf.ones((2, 2))

y = my_sum(x)
print(y.numpy())  # [2. 2.]

y = my_sum(x)
print(y.numpy())  # [4. 4.]

assert my_sum.weights == [my_sum.total]
assert my_sum.non_trainable_weights == [my_sum.total]
assert my_sum.trainable_weights == []
```

For more information about creating layers, see the guide Making new Layers and Models via subclassing

---

`weights` property

```
tf.keras.layers.Layer.weights
```

Returns the list of all layer variables/weights.

Returns

A list of variables.

`trainable_weights` property

```
tf.keras.layers.Layer.trainable_weights
```

List of all trainable weights tracked by this layer.

Trainable weights are updated via gradient descent during training.

Returns

A list of trainable variables.

`non_trainable_weights` property

```
tf.keras.layers.Layer.non_trainable_weights
```

List of all non-trainable weights tracked by this layer.

Non-trainable weights are *not* updated during training. They are expected to be updated manually in `call()`.

Returns

A list of non-trainable variables.

`add_weight` method

[source]

```
Layer.add_weight(
    name=None,
    shape=None,
    dtype=None,
    initializer=None,
    regularizer=None,
    trainable=None,
    constraint=None,
    use_resource=None,
    synchronization=tf.VariableSynchronization.AUTO,
    aggregation=tf.VariableSynchronization.NONE,
    **kwargs
)
```

Adds a new variable to the layer.

Arguments

- **name**: Variable name.
- **shape**: Variable shape. Defaults to scalar if unspecified.
- **dtype**: The type of the variable. Defaults to `self.dtype`.
- **initializer**: Initializer instance (callable).
- **regularizer**: Regularizer instance (callable).
- **trainable**: Boolean, whether the variable should be part of the layer's "trainable_variables" (e.g. variables, biases) or "non_trainable_variables" (e.g. BatchNorm mean and variance). Note that `trainable` cannot be `True` if `synchronization` is set to `ON_READ`.
- **constraint**: Constraint instance (callable).
- **use_resource**: Whether to use a `ResourceVariable` or not. See this guide for more information.
- **synchronization**: Indicates when a distributed a variable will be aggregated. Accepted values are constants defined in the class `tf.VariableSynchronization`. By default the synchronization is set to `AUTO` and the current `DistributionStrategy` chooses when to synchronize. If `synchronization` is set to `ON_READ`, `trainable` must not be set to `True`.
- **aggregation**: Indicates how a distributed variable will be aggregated. Accepted values are constants defined in the class `tf.VariableAggregation`.
- ****kwargs**: Additional keyword arguments. Accepted values are `getter`, `collections`, `experimental_autocast` and `caching_device`.

Returns

The variable created.

Raises

- **ValueError**: When giving unsupported dtype and no initializer or when trainable has been set to True with synchronization set as `ON_READ`.

`trainable` property

```
tf.keras.layers.Layer.trainable
```

`get_weights` method

[source]

```
Layer.get_weights()
```

Returns the current weights of the layer, as NumPy arrays.

The weights of a layer represent the state of the layer. This function returns both trainable and non-trainable weight values associated with this layer as a list of NumPy arrays, which can in turn be used to load state into similarly parameterized layers.

For example, a `Dense` layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another `Dense` layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
...   kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
...   kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
       [2.],
       [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
```

Returns

Weights values as a list of NumPy arrays.

`set_weights` method

[source]

```
Layer.set_weights(weights)
```

Sets the weights of the layer, from NumPy arrays.

The weights of a layer represent the state of the layer. This function sets the weight values from numpy arrays. The weight values should be passed in the order they are created by the layer. Note that the layer's weights must be instantiated before calling this function, by calling the layer.

For example, a `Dense` layer returns a list of two values: the kernel matrix and the bias vector. These can be used to set the weights of another `Dense` layer:

```
>>> layer_a = tf.keras.layers.Dense(1,
...   kernel_initializer=tf.constant_initializer(1.))
>>> a_out = layer_a(tf.convert_to_tensor([[1., 2., 3.]]))
>>> layer_a.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b = tf.keras.layers.Dense(1,
...   kernel_initializer=tf.constant_initializer(2.))
>>> b_out = layer_b(tf.convert_to_tensor([[10., 20., 30.]]))
>>> layer_b.get_weights()
[array([[2.],
       [2.],
       [2.]], dtype=float32), array([0.], dtype=float32)]
>>> layer_b.set_weights(layer_a.get_weights())
>>> layer_b.get_weights()
[array([[1.],
       [1.],
       [1.]], dtype=float32), array([0.], dtype=float32)]
```

Arguments

- **weights**: a list of NumPy arrays. The number of arrays and their shape must match number of the dimensions of the weights of the layer (i.e. it should match the output of `get_weights`).

Raises

- **ValueError**: If the provided weights list does not match the layer's specifications.

`get_config` method

[source]

```
Model.get_config()
```

Returns the config of the `Model`.

Config is a Python dictionary (serializable) containing the configuration of an object, which in this case is a `Model`. This allows the `Model` to be be reinstantiated later (without its trained weights) from this configuration.

Note that `get_config()` does not guarantee to return a fresh copy of dict every time it is called. The callers should make a copy of the returned dict if they want to modify it.

Developers of subclassed `Model` are advised to override this method, and continue to update the dict from `super(MyModel, self).get_config()` to provide the proper configuration of this `Model`. The default config is an empty dict. Optionally, raise `NotImplementedError` to allow Keras to attempt a default serialization.

Returns

Python dictionary containing the configuration of this `Model`.

`add_loss` method

[source]

```
Layer.add_loss(losses, **kwargs)
```

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 the 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.

This method can be used inside a subclassed layer or model's `call` function, in which case `losses` should be a Tensor or list of Tensors.

Example

```
class MyLayer(tf.keras.layers.Layer):
  def call(self, inputs):
    self.add_loss(tf.abs(tf.reduce_mean(inputs)))
    return inputs
```

This method can also be called directly on a Functional Model during construction. In this case, any loss Tensors passed to this Model must be symbolic and be able to be traced back to the model's `Input`s. These losses become part of the model's topology and are tracked in `get_config`.

Example

```
inputs = tf.keras.Input(shape=(10,))
x = tf.keras.layers.Dense(10)(inputs)
outputs = tf.keras.layers.Dense(1)(x)
model = tf.keras.Model(inputs, outputs)
# Activity regularization.
model.add_loss(tf.abs(tf.reduce_mean(x)))
```

If this is not the case for your loss (if, for example, your loss references a `Variable` of one of the model's layers), you can wrap your loss in a zero-argument lambda. These losses are not tracked as part of the model's topology since they can't be serialized.

Example

```
inputs = tf.keras.Input(shape=(10,))
d = tf.keras.layers.Dense(10)
x = d(inputs)
outputs = tf.keras.layers.Dense(1)(x)
model = tf.keras.Model(inputs, outputs)
# Weight regularization.
model.add_loss(lambda: tf.reduce_mean(d.kernel))
```

Arguments

- **losses**: Loss tensor, or list/tuple of tensors. Rather than tensors, losses may also be zero-

argument callables which create a loss tensor.
- ****kwargs**: Used for backwards compatibility only.

`add_metric` method

[source]

```
Layer.add_metric(value, name=None, **kwargs)
```

Adds metric tensor to the layer.

This method can be used inside the `call()` method of a subclassed layer or model.

```
class MyMetricLayer(tf.keras.layers.Layer):
  def __init__(self):
    super(MyMetricLayer, self).__init__(name='my_metric_layer')
    self.mean = tf.keras.metrics.Mean(name='metric_1')

  def call(self, inputs):
    self.add_metric(self.mean(inputs))
    self.add_metric(tf.reduce_sum(inputs), name='metric_2')
    return inputs
```

This method can also be called directly on a Functional Model during construction. In this case, any tensor passed to this Model must be symbolic and be able to be traced back to the model's `Input`s. These metrics become part of the model's topology and are tracked when you save the model via `save()`.

```
inputs = tf.keras.Input(shape=(10,))
x = tf.keras.layers.Dense(10)(inputs)
outputs = tf.keras.layers.Dense(1)(x)
model = tf.keras.Model(inputs, outputs)
model.add_metric(math_ops.reduce_sum(x), name='metric_1')
```

Note: Calling `add_metric()` with the result of a metric object on a Functional Model, as shown in the example below, is not supported. This is because we cannot trace the metric result tensor back to the model's inputs.

```
inputs = tf.keras.Input(shape=(10,))
x = tf.keras.layers.Dense(10)(inputs)
outputs = tf.keras.layers.Dense(1)(x)
model = tf.keras.Model(inputs, outputs)
model.add_metric(tf.keras.metrics.Mean()(x), name='metric_1')
```

Arguments

- **value**: Metric tensor.
- **name**: String metric name.
- ****kwargs**: Additional keyword arguments for backward compatibility. Accepted values: `aggregation` - When the `value` tensor provided is not the result of calling a `keras.Metric` instance, it will be aggregated by default using a `keras.Metric.Mean`.

`losses` property

```
tf.keras.layers.Layer.losses
```

List of losses added using the `add_loss()` API.

Variable regularization tensors are created when this property is accessed, so it is eager safe: accessing `losses` under a `tf.GradientTape` will propagate gradients back to the corresponding variables.

Examples

```
>>> class MyLayer(tf.keras.layers.Layer):
...   def call(self, inputs):
...     self.add_loss(tf.abs(tf.reduce_mean(inputs)))
...     return inputs
>>> l = MyLayer()
>>> l(np.ones((10, 1)))
>>> l.losses
[1.0]
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> x = tf.keras.layers.Dense(10)(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Activity regularization.
>>> len(model.losses)
0
>>> model.add_loss(tf.abs(tf.reduce_mean(x)))
>>> len(model.losses)
1
```

```
>>> inputs = tf.keras.Input(shape=(10,))
>>> d = tf.keras.layers.Dense(10, kernel_initializer='ones')
>>> x = d(inputs)
>>> outputs = tf.keras.layers.Dense(1)(x)
>>> model = tf.keras.Model(inputs, outputs)
>>> # Weight regularization.
>>> model.add_loss(lambda: tf.reduce_mean(d.kernel))
>>> model.losses
[<tf.Tensor: shape=(), dtype=float32, numpy=1.0>]
```

Returns

A list of tensors.

`metrics` property

```
tf.keras.layers.Layer.metrics
```

List of metrics added using the `add_metric()` API.

Example

```
>>> input = tf.keras.layers.Input(shape=(3,))
>>> d = tf.keras.layers.Dense(2)
>>> output = d(input)
>>> d.add_metric(tf.reduce_max(output), name='max')
>>> d.add_metric(tf.reduce_min(output), name='min')
>>> [m.name for m in d.metrics]
['max', 'min']
```

Returns

A list of `Metric` objects.

`dynamic` property

```
tf.keras.layers.Layer.dynamic
```

Whether the layer is dynamic (eager-only); set in the constructor.

The base Layer class