Lambda layer

Lambda class

```
tf.keras.layers.Lambda(
    function, output_shape=None, mask=None, arguments=None, **kwargs
)
```

Wraps arbitrary expressions as a `Layer` object.

The `Lambda` layer exists so that arbitrary expressions can be used as a `Layer` when constructing `Sequential` and Functional API models. `Lambda` layers are best suited for simple operations or quick experimentation. For more advanced use cases, follow this guide for subclassing `tf.keras.layers.Layer`.

WARNING: `tf.keras.layers.Lambda` layers have (de)serialization limitations!

The main reason to subclass `tf.keras.layers.Layer` instead of using a `Lambda` layer is saving and inspecting a Model. `Lambda` layers are saved by serializing the Python bytecode, which is fundamentally non-portable. They should only be loaded in the same environment where they were saved. Subclassed layers can be saved in a more portable way by overriding their `get_config` method. Models that rely on subclassed Layers are also often easier to visualize and reason about.

Examples

```
# add a x -> x^2 layer
model.add(Lambda(lambda x: x ** 2))
```

```
# add a layer that returns the concatenation
# of the positive part of the input and
# the opposite of the negative part

def antirectifier(x):
    x -= K.mean(x, axis=1, keepdims=True)
    x = K.l2_normalize(x, axis=1)
    pos = K.relu(x)
    neg = K.relu(-x)
    return K.concatenate([pos, neg], axis=1)

model.add(Lambda(antirectifier))
```

Variables: While it is possible to use Variables with Lambda layers, this practice is discouraged as it can easily lead to bugs. For instance, consider the following layer:

`python scale = tf.Variable(1.) scale_layer = tf.keras.layers.Lambda(lambda x: x * scale)`

Because scale_layer does not directly track the `scale` variable, it will not appear in `scale_layer.trainable_weights` and will therefore not be trained if `scale_layer` is used in a Model.

A better pattern is to write a subclassed Layer:

```python class ScaleLayer(tf.keras.layers.Layer): def **init**(self): super(ScaleLayer, self).**init**() self.scale = tf.Variable(1.)

```
  def call(self, inputs):
    return inputs * self.scale
```

```

In general, Lambda layers can be convenient for simple stateless computation, but anything more complex should use a subclass Layer instead.

Arguments

- **function**: The function to be evaluated. Takes input tensor as first argument.

- **output_shape**: Expected output shape from function. This argument can be inferred if not explicitly provided. Can be a tuple or function. If a tuple, it only specifies the first dimension onward; sample dimension is assumed either the same as the input: `output_shape = (input_shape[0], ) + output_shape` or, the input is `None` and the sample dimension is also `None`: `output_shape = (None, ) + output_shape` If a function, it specifies the entire shape as a function of the input shape: `output_shape = f(input_shape)`
- **mask**: Either None (indicating no masking) or a callable with the same signature as the `compute_mask` layer method, or a tensor that will be returned as output mask regardless of what the input is.
- **arguments**: Optional dictionary of keyword arguments to be passed to the function.

Input shape

Arbitrary. Use the keyword argument input_shape (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

Output shape

Specified by `output_shape` argument