from \_\_future\_\_ import absolute\_import

from \_\_future\_\_ import division

from \_\_future\_\_ import print\_function

import tensorflow as tf

from tensorflow.python.eager import context

from tensorflow.python.framework import device as tfdev

from tensorflow.python.framework import ops as tf\_ops

from tensorflow.python.ops import image\_ops as tf\_image\_ops

from tensorflow.python.ops import math\_ops as tf\_math\_ops

from tensorflow.python.ops import state\_ops as tf\_state\_ops

from tensorflow.python.keras import backend as tf\_keras\_backend

from tensorflow.python.keras.utils import tf\_utils

from tensorflow.python.ops import functional\_ops

from tensorflow.python.ops import ctc\_ops as ctc

from .common import floatx, epsilon, image\_data\_format

import sys

import functools

import threading

import numpy as np

from distutils.version import StrictVersion

from ..utils.generic\_utils import transpose\_shape

py\_all = all

py\_any = any

py\_sum = sum

py\_slice = slice

# INTERNAL UTILS

# This list holds the available devices.

# It is populated when `\_get\_available\_gpus()` is called for the first time.

# We assume our devices don't change during our lifetime.

\_LOCAL\_DEVICES = None

\_SYMBOLIC\_SCOPE = threading.local()

\_SYMBOLIC\_SCOPE.value = True

\_LEARNING\_PHASE\_CACHE = {}

def \_is\_tf\_1():

return tf.\_\_version\_\_.startswith('1.')

# Set initial config

tf\_keras\_backend.set\_floatx(floatx())

tf\_keras\_backend.set\_epsilon(epsilon())

tf\_keras\_backend.set\_image\_data\_format(image\_data\_format())

# Private TF Keras utils

get\_graph = tf\_keras\_backend.get\_graph

# learning\_phase\_scope = tf\_keras\_backend.learning\_phase\_scope # TODO

name\_scope = tf.name\_scope

def symbolic(func):

"""Decorator used in TensorFlow 2.0 to enter the Keras graph.

# Arguments

func: Function to decorate.

# Returns

Decorated function.

"""

if \_is\_tf\_1():

return func

@functools.wraps(func)

def symbolic\_fn\_wrapper(\*args, \*\*kwargs):

if \_SYMBOLIC\_SCOPE.value:

with get\_graph().as\_default():

return func(\*args, \*\*kwargs)

else:

return func(\*args, \*\*kwargs)

return symbolic\_fn\_wrapper

def is\_symbolic(x):

return isinstance(x, tf.Tensor) and hasattr(x, 'op')

def eager(func):

"""Decorator used in TensorFlow 2.0 to exit the Keras graph.

# Arguments

func: Function to decorate.

# Returns

Decorated function.

"""

if \_is\_tf\_1():

return func

global \_SYMBOLIC\_SCOPE

@functools.wraps(func)

def eager\_fn\_wrapper(\*args, \*\*kwargs):

prev\_value = \_SYMBOLIC\_SCOPE.value

try:

\_SYMBOLIC\_SCOPE.value = False

with context.eager\_mode():

out = func(\*args, \*\*kwargs)

finally:

\_SYMBOLIC\_SCOPE.value = prev\_value

return out

return eager\_fn\_wrapper

def \_has\_compat\_v1():

if hasattr(tf, 'compat') and hasattr(tf.compat, 'v1'):

return True

return False

def get\_uid(prefix=''):

"""Provides a unique UID given a string prefix.

# Arguments

prefix: string.

# Returns

An integer.

# Example

```python

>>> keras.backend.get\_uid('dense')

1

>>> keras.backend.get\_uid('dense')

2

```

"""

return tf\_keras\_backend.get\_uid(prefix)

def manual\_variable\_initialization(value):

"""Sets the manual variable initialization flag.

This boolean flag determines whether

variables should be initialized

as they are instantiated (default), or if

the user should handle the initialization.

# Arguments

value: Python boolean.

"""

tf\_keras\_backend.manual\_variable\_initialization(value)

def epsilon():

"""Returns the value of the fuzz factor used in numeric expressions.

# Returns

A float.

# Example

```python

>>> keras.backend.epsilon()

1e-07

```

"""

return tf\_keras\_backend.epsilon()

def reset\_uids():

"""Resets graph identifiers."""

tf\_keras\_backend.reset\_uids()

def set\_epsilon(e):

"""Sets the value of the fuzz factor used in numeric expressions.

# Arguments

e: float. New value of epsilon.

# Example

```python

>>> from keras import backend as K

>>> K.epsilon()

1e-07

>>> K.set\_epsilon(1e-05)

>>> K.epsilon()

1e-05

```

"""

tf\_keras\_backend.set\_epsilon(e)

def floatx():

"""Returns the default float type, as a string.

(e.g. 'float16', 'float32', 'float64').

# Returns

String, the current default float type.

# Example

```python

>>> keras.backend.floatx()

'float32'

```

"""

return tf\_keras\_backend.floatx()

def set\_floatx(floatx):

"""Sets the default float type.

# Arguments

floatx: String, 'float16', 'float32', or 'float64'.

# Example

```python

>>> from keras import backend as K

>>> K.floatx()

'float32'

>>> K.set\_floatx('float16')

>>> K.floatx()

'float16'

```

"""

tf\_keras\_backend.set\_floatx(floatx)

def cast\_to\_floatx(x):

"""Cast a Numpy array to the default Keras float type.

# Arguments

x: Numpy array.

# Returns

The same Numpy array, cast to its new type.

# Example

```python

>>> from keras import backend as K

>>> K.floatx()

'float32'

>>> arr = numpy.array([1.0, 2.0], dtype='float64')

>>> arr.dtype

dtype('float64')

>>> new\_arr = K.cast\_to\_floatx(arr)

>>> new\_arr

array([ 1., 2.], dtype=float32)

>>> new\_arr.dtype

dtype('float32')

```

"""

return tf\_keras\_backend.cast\_to\_floatx(x)

def image\_data\_format():

"""Returns the default image data format convention.

# Returns

A string, either `'channels\_first'` or `'channels\_last'`

# Example

```python

>>> keras.backend.image\_data\_format()

'channels\_first'

```

"""

return tf\_keras\_backend.image\_data\_format()

def set\_image\_data\_format(data\_format):

"""Sets the value of the data format convention.

# Arguments

data\_format: string. `'channels\_first'` or `'channels\_last'`.

# Example

```python

>>> from keras import backend as K

>>> K.image\_data\_format()

'channels\_first'

>>> K.set\_image\_data\_format('channels\_last')

>>> K.image\_data\_format()

'channels\_last'

```

"""

tf\_keras\_backend.set\_image\_data\_format(data\_format)

def normalize\_data\_format(value):

"""Checks that the value correspond to a valid data format.

# Arguments

value: String or None. `'channels\_first'` or `'channels\_last'`.

# Returns

A string, either `'channels\_first'` or `'channels\_last'`

# Example

```python

>>> from keras import backend as K

>>> K.normalize\_data\_format(None)

'channels\_first'

>>> K.normalize\_data\_format('channels\_last')

'channels\_last'

```

# Raises

ValueError: if `value` or the global `data\_format` invalid.

"""

if value is None:

value = image\_data\_format()

data\_format = value.lower()

if data\_format not in {'channels\_first', 'channels\_last'}:

raise ValueError('The `data\_format` argument must be one of '

'"channels\_first", "channels\_last". Received: ' +

str(value))

return data\_format

@symbolic

def learning\_phase():

"""Returns the learning phase flag.

The learning phase flag is a bool tensor (0 = test, 1 = train)

to be passed as input to any Keras function

that uses a different behavior at train time and test time.

# Returns

Learning phase (scalar integer tensor or Python integer).

"""

lp = tf\_keras\_backend.learning\_phase()

if \_is\_tf\_1():

return lp

else:

if isinstance(lp, int):

return lp

if id(lp) in \_LEARNING\_PHASE\_CACHE:

return \_LEARNING\_PHASE\_CACHE[id(lp)]

with name\_scope(''):

int\_lp = tf.cast(lp, 'int32', name='learning\_phase')

\_LEARNING\_PHASE\_CACHE[id(lp)] = int\_lp

return int\_lp

@symbolic

def set\_learning\_phase(value):

"""Sets the learning phase to a fixed value.

# Arguments

value: Learning phase value, either 0 or 1 (integers).

# Raises

ValueError: if `value` is neither `0` nor `1`.

"""

tf\_keras\_backend.set\_learning\_phase(value)

def get\_session():

"""Returns the TF session to be used by the backend.

If a default TensorFlow session is available, we will return it.

Else, we will return the global Keras session.

If no global Keras session exists at this point:

we will create a new global session.

Note that you can manually set the global session

via `K.set\_session(sess)`.

# Returns

A TensorFlow session.

# Raises

RuntimeError: if no session is available

(e.g. when using TensorFlow 2.0).

"""

if not \_is\_tf\_1():

raise RuntimeError(

'`get\_session` is not available '

'when using TensorFlow 2.0.')

if tf.executing\_eagerly():

raise RuntimeError(

'`get\_session` is not available when '

'TensorFlow is executing eagerly.')

return tf\_keras\_backend.get\_session()

def set\_session(session):

"""Sets the global TensorFlow session.

# Arguments

session: A TF Session.

# Raises

RuntimeError: if no session is available

(e.g. when using TensorFlow 2.0).

"""

if not \_is\_tf\_1():

raise RuntimeError(

'`set\_session` is not available '

'when using TensorFlow 2.0.')

if tf.executing\_eagerly():

raise RuntimeError(

'`set\_session` is not available when '

'TensorFlow is executing eagerly.')

tf\_keras\_backend.set\_session(session)

def clear\_session():

"""Destroys the current Keras graph and creates a new one.

Useful to avoid clutter from old models / layers.

"""

tf\_keras\_backend.clear\_session()

global \_LEARNING\_PHASE\_CACHE

\_LEARNING\_PHASE\_CACHE = {}

def v1\_variable\_initialization():

session = get\_session()

with session.graph.as\_default():

variables = tf.global\_variables()

candidate\_vars = []

for v in variables:

if not getattr(v, '\_keras\_initialized', False):

candidate\_vars.append(v)

if candidate\_vars:

# This step is expensive, so we only run it on variables

# not already marked as initialized.

is\_initialized = session.run(

[tf.is\_variable\_initialized(v) for v in candidate\_vars])

uninitialized\_vars = []

for flag, v in zip(is\_initialized, candidate\_vars):

if not flag:

uninitialized\_vars.append(v)

v.\_keras\_initialized = True

if uninitialized\_vars:

session.run(tf.variables\_initializer(uninitialized\_vars))

# DEVICE MANIPULATION AND PROBING

class \_TfDeviceCaptureOp(object):

"""Class for capturing the TF device scope."""

def \_\_init\_\_(self):

# NOTE(robieta): This differs from tf.keras in that self.device is a

# DeviceSpec rather than a string. This is done for compatibility

# with a range of TensorFlow versions.

self.device = None

def \_set\_device(self, device):

"""This method captures TF's explicit device scope setting."""

self.device = device

def \_set\_device\_from\_string(self, device\_str):

self.device = tfdev.DeviceSpec.from\_string(device\_str)

def \_get\_current\_tf\_device():

"""Return explicit device of current context, otherwise returns `None`.

# Returns

If the current device scope is explicitly set, it returns a string with

the device (`CPU` or `GPU`). If the scope is not explicitly set, it will

return `None`.

"""

g = get\_graph()

op = \_TfDeviceCaptureOp()

g.\_apply\_device\_functions(op)

return op.device

def \_is\_current\_explicit\_device(device\_type):

"""Check if the current device is explicitly set on the device type specified.

# Arguments

device\_type: A string containing `GPU` or `CPU` (case-insensitive).

# Returns

A boolean indicating if the current device

scope is explicitly set on the device type.

# Raises

ValueError: If the `device\_type` string indicates an unsupported device.

"""

device\_type = device\_type.lower()

if device\_type not in ['cpu', 'gpu']:

raise ValueError('`device\_type` should be either "cpu" or "gpu".')

device = \_get\_current\_tf\_device()

return (device is not None and device.device\_type.lower() == device\_type)

def \_get\_available\_gpus():

"""Get a list of available gpu devices (formatted as strings).

# Returns

A list of available GPU devices.

"""

global \_LOCAL\_DEVICES

if \_LOCAL\_DEVICES is None:

if \_is\_tf\_1():

devices = get\_session().list\_devices()

\_LOCAL\_DEVICES = [x.name for x in devices]

else:

\_LOCAL\_DEVICES = tf.config.experimental\_list\_devices()

return [x for x in \_LOCAL\_DEVICES if 'device:gpu' in x.lower()]

def \_has\_nchw\_support():

"""Check whether the current scope supports NCHW ops.

TensorFlow does not support NCHW on CPU.

Therefore we check if we are not explicitly put on

CPU, and have GPUs available.

In this case there will be soft-placing on the GPU device.

# Returns

bool: if the current scope device placement would support nchw

"""

explicitly\_on\_cpu = \_is\_current\_explicit\_device('cpu')

gpus\_available = len(\_get\_available\_gpus()) > 0

return (not explicitly\_on\_cpu and gpus\_available)

# VARIABLE MANIPULATION

@symbolic

def \_to\_tensor(x, dtype):

"""Convert the input `x` to a tensor of type `dtype`.

# Arguments

x: An object to be converted (numpy array, list, tensors).

dtype: The destination type.

# Returns

A tensor.

"""

return tf.convert\_to\_tensor(x, dtype=dtype)

def is\_sparse(tensor):

"""Returns whether a tensor is a sparse tensor.

# Arguments

tensor: A tensor instance.

# Returns

A boolean.

# Example

```python

>>> from keras import backend as K

>>> a = K.placeholder((2, 2), sparse=False)

>>> print(K.is\_sparse(a))

False

>>> b = K.placeholder((2, 2), sparse=True)

>>> print(K.is\_sparse(b))

True

```

"""

return isinstance(tensor, tf.SparseTensor)

@symbolic

def to\_dense(tensor):

"""Converts a sparse tensor into a dense tensor and returns it.

# Arguments

tensor: A tensor instance (potentially sparse).

# Returns

A dense tensor.

# Examples

```python

>>> from keras import backend as K

>>> b = K.placeholder((2, 2), sparse=True)

>>> print(K.is\_sparse(b))

True

>>> c = K.to\_dense(b)

>>> print(K.is\_sparse(c))

False

```

"""

if is\_sparse(tensor):

return tf.sparse.to\_dense(tensor)

else:

return tensor

def variable(value, dtype=None, name=None, constraint=None):

"""Instantiates a variable and returns it.

# Arguments

value: Numpy array, initial value of the tensor.

dtype: Tensor type.

name: Optional name string for the tensor.

constraint: Optional projection function to be

applied to the variable after an optimizer update.

# Returns

A variable instance (with Keras metadata included).

# Examples

```python

>>> from keras import backend as K

>>> val = np.array([[1, 2], [3, 4]])

>>> kvar = K.variable(value=val, dtype='float64', name='example\_var')

>>> K.dtype(kvar)

'float64'

>>> print(kvar)

example\_var

>>> K.eval(kvar)

array([[ 1., 2.],

[ 3., 4.]])

```

"""

v = tf\_keras\_backend.variable(

value, dtype=dtype, name=name, constraint=constraint)

if hasattr(value, 'tocoo'):

v.\_keras\_shape = value.tocoo().shape

elif isinstance(value, np.ndarray):

v.\_keras\_shape = value.shape

elif hasattr(value, 'shape'):

v.\_keras\_shape = int\_shape(value)

v.\_uses\_learning\_phase = False

return v

def is\_variable(x):

return isinstance(x, tf.Variable)

def constant(value, dtype=None, shape=None, name=None):

"""Creates a constant tensor.

# Arguments

value: A constant value (or list)

dtype: The type of the elements of the resulting tensor.

shape: Optional dimensions of resulting tensor.

name: Optional name for the tensor.

# Returns

A Constant Tensor.

"""

with tf\_ops.init\_scope():

return tf\_keras\_backend.constant(

value, dtype=dtype, shape=shape, name=name)

def is\_keras\_tensor(x):

"""Returns whether `x` is a Keras tensor.

A "Keras tensor" is a tensor that was returned by a Keras layer,

(`Layer` class) or by `Input`.

# Arguments

x: A candidate tensor.

# Returns

A boolean: Whether the argument is a Keras tensor.

# Raises

ValueError: In case `x` is not a symbolic tensor.

# Examples

```python

>>> from keras import backend as K

>>> from keras.layers import Input, Dense

>>> np\_var = numpy.array([1, 2])

>>> K.is\_keras\_tensor(np\_var) # A numpy array is not a symbolic tensor.

ValueError

>>> k\_var = tf.placeholder('float32', shape=(1,1))

>>> # A variable indirectly created outside of keras is not a Keras tensor.

>>> K.is\_keras\_tensor(k\_var)

False

>>> keras\_var = K.variable(np\_var)

>>> # A variable created with the keras backend is not a Keras tensor.

>>> K.is\_keras\_tensor(keras\_var)

False

>>> keras\_placeholder = K.placeholder(shape=(2, 4, 5))

>>> # A placeholder is not a Keras tensor.

>>> K.is\_keras\_tensor(keras\_placeholder)

False

>>> keras\_input = Input([10])

>>> K.is\_keras\_tensor(keras\_input) # An Input is a Keras tensor.

True

>>> keras\_layer\_output = Dense(10)(keras\_input)

>>> # Any Keras layer output is a Keras tensor.

>>> K.is\_keras\_tensor(keras\_layer\_output)

True

```

"""

if not is\_tensor(x):

raise ValueError('Unexpectedly found an instance of type `' +

str(type(x)) + '`. '

'Expected a symbolic tensor instance.')

return hasattr(x, '\_keras\_history')

def is\_tensor(x):

return isinstance(x, tf\_ops.\_TensorLike) or tf\_ops.is\_dense\_tensor\_like(x)

@symbolic

def placeholder(shape=None, ndim=None, dtype=None, sparse=False, name=None):

"""Instantiates a placeholder tensor and returns it.

# Arguments

shape: Shape of the placeholder

(integer tuple, may include `None` entries).

ndim: Number of axes of the tensor.

At least one of {`shape`, `ndim`} must be specified.

If both are specified, `shape` is used.

dtype: Placeholder type.

sparse: Boolean, whether the placeholder should have a sparse type.

name: Optional name string for the placeholder.

# Returns

Tensor instance (with Keras metadata included).

# Examples

```python

>>> from keras import backend as K

>>> input\_ph = K.placeholder(shape=(2, 4, 5))

>>> input\_ph.\_keras\_shape

(2, 4, 5)

>>> input\_ph

<tf.Tensor 'Placeholder\_4:0' shape=(2, 4, 5) dtype=float32>

```

"""

if dtype is None:

dtype = floatx()

x = tf\_keras\_backend.placeholder(

shape=shape, ndim=ndim, dtype=dtype, sparse=sparse, name=name)

if shape is None:

if ndim is not None:

shape = tuple(None for \_ in range(ndim))

x.\_keras\_shape = shape

x.\_uses\_learning\_phase = False

return x

@symbolic

def is\_placeholder(x):

"""Returns whether `x` is a placeholder.

# Arguments

x: A candidate placeholder.

# Returns

Boolean.

"""

try:

return x.op.type == 'Placeholder'

except AttributeError:

return False

def shape(x):

"""Returns the symbolic shape of a tensor or variable.

# Arguments

x: A tensor or variable.

# Returns

A symbolic shape (which is itself a tensor).

# Examples

```python

# TensorFlow example

>>> from keras import backend as K

>>> tf\_session = K.get\_session()

>>> val = np.array([[1, 2], [3, 4]])

>>> kvar = K.variable(value=val)

>>> inputs = keras.backend.placeholder(shape=(2, 4, 5))

>>> K.shape(kvar)

<tf.Tensor 'Shape\_8:0' shape=(2,) dtype=int32>

>>> K.shape(inputs)

<tf.Tensor 'Shape\_9:0' shape=(3,) dtype=int32>

# To get integer shape (Instead, you can use K.int\_shape(x))

>>> K.shape(kvar).eval(session=tf\_session)

array([2, 2], dtype=int32)

>>> K.shape(inputs).eval(session=tf\_session)

array([2, 4, 5], dtype=int32)

```

"""

return tf.shape(x)

def int\_shape(x):

"""Returns the shape of tensor or variable as a tuple of int or None entries.

# Arguments

x: Tensor or variable.

# Returns

A tuple of integers (or None entries).

# Examples

```python

>>> from keras import backend as K

>>> inputs = K.placeholder(shape=(2, 4, 5))

>>> K.int\_shape(inputs)

(2, 4, 5)

>>> val = np.array([[1, 2], [3, 4]])

>>> kvar = K.variable(value=val)

>>> K.int\_shape(kvar)

(2, 2)

```

{{np\_implementation}}

"""

if hasattr(x, '\_keras\_shape'):

return x.\_keras\_shape

try:

if isinstance(x.shape, tuple):

return x.shape

return tuple(x.shape.as\_list())

except ValueError:

return None

def ndim(x):

"""Returns the number of axes in a tensor, as an integer.

# Arguments

x: Tensor or variable.

# Returns

Integer (scalar), number of axes.

# Examples

```python

>>> from keras import backend as K

>>> inputs = K.placeholder(shape=(2, 4, 5))

>>> val = np.array([[1, 2], [3, 4]])

>>> kvar = K.variable(value=val)

>>> K.ndim(inputs)

3

>>> K.ndim(kvar)

2

```

{{np\_implementation}}

"""

return x.shape.rank

def size(x, name=None):

"""Returns the size of a tensor.

# Arguments

x: Tensor or variable.

name: A name for the operation (optional).

# Returns

Size of the tensor.

# Examples

```python

>>> from keras import backend as K

>>> val = np.array([[1, 2], [3, 4]])

>>> kvar = K.variable(value=val)

>>> K.size(inputs)

<tf.Tensor: id=9, shape=(), dtype=int32, numpy=4>

```

"""

if is\_symbolic(x):

with get\_graph().as\_default():

return tf.size(x)

return tf.size(x, name=name)

def dtype(x):

"""Returns the dtype of a Keras tensor or variable, as a string.

# Arguments

x: Tensor or variable.

# Returns

String, dtype of `x`.

# Examples

```python

>>> from keras import backend as K

>>> K.dtype(K.placeholder(shape=(2,4,5)))

'float32'

>>> K.dtype(K.placeholder(shape=(2,4,5), dtype='float32'))

'float32'

>>> K.dtype(K.placeholder(shape=(2,4,5), dtype='float64'))

'float64'

# Keras variable

>>> kvar = K.variable(np.array([[1, 2], [3, 4]]))

>>> K.dtype(kvar)

'float32\_ref'

>>> kvar = K.variable(np.array([[1, 2], [3, 4]]), dtype='float32')

>>> K.dtype(kvar)

'float32\_ref'

```

{{np\_implementation}}

"""

return x.dtype.base\_dtype.name

def eval(x):

"""Evaluates the value of a tensor.

# Arguments

x: A tensor.

# Returns

A Numpy array.

# Examples

```python

>>> from keras import backend as K

>>> kvar = K.variable(np.array([[1, 2], [3, 4]]), dtype='float32')

>>> K.eval(kvar)

array([[ 1., 2.],

[ 3., 4.]], dtype=float32)

```

{{np\_implementation}}

"""

if \_is\_tf\_1():

return to\_dense(x).eval(session=get\_session())

if hasattr(x, 'numpy'):

with context.eager\_mode():

return x.numpy()

eval\_fn = function([], [x])

return eval\_fn([])[0]

def zeros(shape, dtype=None, name=None):

"""Instantiates an all-zeros variable and returns it.

# Arguments

shape: Tuple of integers, shape of returned Keras variable

dtype: String, data type of returned Keras variable

name: String, name of returned Keras variable

# Returns

A variable (including Keras metadata), filled with `0.0`.

Note that if `shape` was symbolic, we cannot return a variable,

and will return a dynamically-shaped tensor instead.

# Example

```python

>>> from keras import backend as K

>>> kvar = K.zeros((3,4))

>>> K.eval(kvar)

array([[ 0., 0., 0., 0.],

[ 0., 0., 0., 0.],

[ 0., 0., 0., 0.]], dtype=float32)

```

{{np\_implementation}}

"""

if dtype is None:

dtype = floatx()

with tf\_ops.init\_scope():

v = tf.zeros(shape=shape, dtype=dtype, name=name)

if py\_all(v.shape.as\_list()):

return variable(v, dtype=dtype, name=name)

return v

def ones(shape, dtype=None, name=None):

"""Instantiates an all-ones variable and returns it.

# Arguments

shape: Tuple of integers, shape of returned Keras variable.

dtype: String, data type of returned Keras variable.

name: String, name of returned Keras variable.

# Returns

A Keras variable, filled with `1.0`.

Note that if `shape` was symbolic, we cannot return a variable,

and will return a dynamically-shaped tensor instead.

# Example

```python

>>> from keras import backend as K

>>> kvar = K.ones((3,4))

>>> K.eval(kvar)

array([[ 1., 1., 1., 1.],

[ 1., 1., 1., 1.],

[ 1., 1., 1., 1.]], dtype=float32)

```

{{np\_implementation}}

"""

if dtype is None:

dtype = floatx()

with tf\_ops.init\_scope():

v = tf.ones(shape=shape, dtype=dtype, name=name)

if py\_all(v.shape.as\_list()):

return variable(v, dtype=dtype, name=name)

return v

def eye(size, dtype=None, name=None):

"""Instantiate an identity matrix and returns it.

# Arguments

size: Tuple, number of rows and columns. If Integer, number of rows.

dtype: String, data type of returned Keras variable.

name: String, name of returned Keras variable.

# Returns

A Keras variable, an identity matrix.

# Example

```python

>>> from keras import backend as K

>>> K.eval(K.eye(3))

array([[ 1., 0., 0.],

[ 0., 1., 0.],

[ 0., 0., 1.]], dtype=float32)

>>> K.eval(K.eye((2, 3)))

array([[1., 0., 0.],

[0., 1., 0.]], dtype=float32)

```

{{np\_implementation}}

"""

if dtype is None:

dtype = floatx()

if isinstance(size, (list, tuple)):

n, m = size

else:

n, m = size, size

with tf\_ops.init\_scope():

return tf.eye(n, m, dtype=dtype, name=name)

@symbolic

def zeros\_like(x, dtype=None, name=None):

"""Instantiates an all-zeros variable of the same shape as another tensor.

# Arguments

x: Keras variable or Keras tensor.

dtype: String, dtype of returned Keras variable.

None uses the dtype of x.

name: String, name for the variable to create.

# Returns

A Keras variable with the shape of x filled with zeros.

# Example

```python

>>> from keras import backend as K

>>> kvar = K.variable(np.random.random((2,3)))

>>> kvar\_zeros = K.zeros\_like(kvar)

>>> K.eval(kvar\_zeros)

array([[ 0., 0., 0.],

[ 0., 0., 0.]], dtype=float32)

```

{{np\_implementation}}

"""

if dtype is None:

dtype = floatx()

return tf.zeros\_like(x, dtype=dtype, name=name)

@symbolic

def ones\_like(x, dtype=None, name=None):

"""Instantiates an all-ones variable of the same shape as another tensor.

# Arguments

x: Keras variable or tensor.

dtype: String, dtype of returned Keras variable.

None uses the dtype of x.

name: String, name for the variable to create.

# Returns

A Keras variable with the shape of x filled with ones.

# Example

```python

>>> from keras import backend as K

>>> kvar = K.variable(np.random.random((2,3)))

>>> kvar\_ones = K.ones\_like(kvar)

>>> K.eval(kvar\_ones)

array([[ 1., 1., 1.],

[ 1., 1., 1.]], dtype=float32)

```

{{np\_implementation}}

"""

if dtype is None:

dtype = floatx()

return tf.ones\_like(x, dtype=dtype, name=name)

@symbolic

def identity(x, name=None):

"""Returns a tensor with the same content as the input tensor.

# Arguments

x: The input tensor.

name: String, name for the variable to create.

# Returns

A tensor of the same shape, type and content.

"""

return tf.identity(x, name)

def random\_uniform\_variable(shape, low, high,

dtype=None,

name=None,

seed=None):

"""Instantiates a variable with values drawn from a uniform distribution.

# Arguments

shape: Tuple of integers, shape of returned Keras variable.

low: Float, lower boundary of the output interval.

high: Float, upper boundary of the output interval.

dtype: String, dtype of returned Keras variable.

name: String, name of returned Keras variable.

seed: Integer, random seed.

# Returns

A Keras variable, filled with drawn samples.

# Example

```python

# TensorFlow example

>>> kvar = K.random\_uniform\_variable((2,3), 0, 1)

>>> kvar

<tensorflow.python.ops.variables.Variable object at 0x10ab40b10>

>>> K.eval(kvar)

array([[ 0.10940075, 0.10047495, 0.476143 ],

[ 0.66137183, 0.00869417, 0.89220798]], dtype=float32)

```

{{np\_implementation}}

"""

if dtype is None:

dtype = floatx()

if seed is None:

# ensure that randomness is conditioned by the Numpy RNG

seed = np.random.randint(10e8)

with tf\_ops.init\_scope():

value = tf.random\_uniform\_initializer(

low, high, seed=seed)(shape, dtype=dtype)

return variable(value, dtype=dtype, name=name)

def random\_normal\_variable(shape, mean, scale, dtype=None,

name=None, seed=None):

"""Instantiates a variable with values drawn from a normal distribution.

# Arguments

shape: Tuple of integers, shape of returned Keras variable.

mean: Float, mean of the normal distribution.

scale: Float, standard deviation of the normal distribution.

dtype: String, dtype of returned Keras variable.

name: String, name of returned Keras variable.

seed: Integer, random seed.

# Returns

A Keras variable, filled with drawn samples.

# Example

```python

# TensorFlow example

>>> kvar = K.random\_normal\_variable((2,3), 0, 1)

>>> kvar

<tensorflow.python.ops.variables.Variable object at 0x10ab12dd0>

>>> K.eval(kvar)

array([[ 1.19591331, 0.68685907, -0.63814116],

[ 0.92629528, 0.28055015, 1.70484698]], dtype=float32)

```

{{np\_implementation}}

"""

if dtype is None:

dtype = floatx()

if seed is None:

# ensure that randomness is conditioned by the Numpy RNG

seed = np.random.randint(10e8)

with tf\_ops.init\_scope():

value = tf.random\_normal\_initializer(

mean, scale, seed=seed)(shape, dtype=dtype)

return variable(value, dtype=dtype, name=name)

def count\_params(x):

"""Returns the static number of elements in a Keras variable or tensor.

# Arguments

x: Keras variable or tensor.

# Returns

Integer, the number of elements in `x`, i.e., the product of the

array's static dimensions.

# Example

```python

>>> kvar = K.zeros((2,3))

>>> K.count\_params(kvar)

6

>>> K.eval(kvar)

array([[ 0., 0., 0.],

[ 0., 0., 0.]], dtype=float32)

```

{{np\_implementation}}

"""

return np.prod(int\_shape(x))

def cast(x, dtype):

"""Casts a tensor to a different dtype and returns it.

You can cast a Keras variable but it still returns a Keras tensor.

# Arguments

x: Keras tensor (or variable).

dtype: String, either (`'float16'`, `'float32'`, or `'float64'`).

# Returns

Keras tensor with dtype `dtype`.

# Example

```python

>>> from keras import backend as K

>>> input = K.placeholder((2, 3), dtype='float32')

>>> input

<tf.Tensor 'Placeholder\_2:0' shape=(2, 3) dtype=float32>

# It doesn't work in-place as below.

>>> K.cast(input, dtype='float16')

<tf.Tensor 'Cast\_1:0' shape=(2, 3) dtype=float16>

>>> input

<tf.Tensor 'Placeholder\_2:0' shape=(2, 3) dtype=float32>

# you need to assign it.

>>> input = K.cast(input, dtype='float16')

>>> input

<tf.Tensor 'Cast\_2:0' shape=(2, 3) dtype=float16>

```

"""

return tf.cast(x, dtype)

# UPDATES OPS

def update(x, new\_x):

"""Update the value of `x` to `new\_x`.

# Arguments

x: A `Variable`.

new\_x: A tensor of same shape as `x`.

# Returns

The variable `x` updated.

"""

return tf\_state\_ops.assign(x, new\_x)

def update\_add(x, increment):

"""Update the value of `x` by adding `increment`.

# Arguments

x: A `Variable`.

increment: A tensor of same shape as `x`.

# Returns

The variable `x` updated.

"""

return tf\_state\_ops.assign\_add(x, increment)

def update\_sub(x, decrement):

"""Update the value of `x` by subtracting `decrement`.

# Arguments

x: A `Variable`.

decrement: A tensor of same shape as `x`.

# Returns

The variable `x` updated.

"""

return tf\_state\_ops.assign\_sub(x, decrement)

@symbolic

def moving\_average\_update(x, value, momentum):

"""Compute the moving average of a variable.

# Arguments

x: A `Variable`.

value: A tensor with the same shape as `x`.

momentum: The moving average momentum.

# Returns

An operation to update the variable.

"""

with tf\_ops.colocate\_with(x):

decay = tf\_ops.convert\_to\_tensor(1.0 - momentum)

if decay.dtype != x.dtype.base\_dtype:

decay = tf\_math\_ops.cast(decay, x.dtype.base\_dtype)

update\_delta = (x - tf\_math\_ops.cast(value, x.dtype)) \* decay

return tf\_state\_ops.assign\_sub(x, update\_delta)

# LINEAR ALGEBRA

def dot(x, y):

"""Multiplies 2 tensors (and/or variables) and returns a \*tensor\*.

When attempting to multiply a nD tensor

with a nD tensor, it reproduces the Theano behavior.

(e.g. `(2, 3) \* (4, 3, 5) -> (2, 4, 5)`)

# Arguments

x: Tensor or variable.

y: Tensor or variable.

# Returns

A tensor, dot product of `x` and `y`.

# Examples

```python

# dot product between tensors

>>> x = K.placeholder(shape=(2, 3))

>>> y = K.placeholder(shape=(3, 4))

>>> xy = K.dot(x, y)

>>> xy

<tf.Tensor 'MatMul\_9:0' shape=(2, 4) dtype=float32>

```

```python

# dot product between tensors

>>> x = K.placeholder(shape=(32, 28, 3))

>>> y = K.placeholder(shape=(3, 4))

>>> xy = K.dot(x, y)

>>> xy

<tf.Tensor 'MatMul\_9:0' shape=(32, 28, 4) dtype=float32>

```

```python

# Theano-like behavior example

>>> x = K.random\_uniform\_variable(shape=(2, 3), low=0, high=1)

>>> y = K.ones((4, 3, 5))

>>> xy = K.dot(x, y)

>>> K.int\_shape(xy)

(2, 4, 5)

```

{{np\_implementation}}

"""

if ndim(x) is not None and (ndim(x) > 2 or ndim(y) > 2):

x\_shape = []

for i, s in zip(int\_shape(x), tf.unstack(tf.shape(x))):

if i is not None:

x\_shape.append(i)

else:

x\_shape.append(s)

x\_shape = tuple(x\_shape)

y\_shape = []

for i, s in zip(int\_shape(y), tf.unstack(tf.shape(y))):

if i is not None:

y\_shape.append(i)

else:

y\_shape.append(s)

y\_shape = tuple(y\_shape)

y\_permute\_dim = list(range(ndim(y)))

y\_permute\_dim = [y\_permute\_dim.pop(-2)] + y\_permute\_dim

xt = tf.reshape(x, [-1, x\_shape[-1]])

yt = tf.reshape(tf.transpose(y, perm=y\_permute\_dim), [y\_shape[-2], -1])

return tf.reshape(tf.matmul(xt, yt),

x\_shape[:-1] + y\_shape[:-2] + y\_shape[-1:])

if is\_sparse(x):

out = tf.sparse.sparse\_dense\_matmul(x, y)

else:

out = tf.matmul(x, y)

return out

def batch\_dot(x, y, axes=None):

"""Batchwise dot product.

`batch\_dot` is used to compute dot product of `x` and `y` when

`x` and `y` are data in batches, i.e. in a shape of

`(batch\_size, :)`.

`batch\_dot` results in a tensor or variable with less dimensions

than the input. If the number of dimensions is reduced to 1,

we use `expand\_dims` to make sure that ndim is at least 2.

# Arguments

x: Keras tensor or variable with `ndim >= 2`.

y: Keras tensor or variable with `ndim >= 2`.

axes: int or tuple(int, int). Target dimensions to be reduced.

# Returns

A tensor with shape equal to the concatenation of `x`'s shape

(less the dimension that was summed over) and `y`'s shape

(less the batch dimension and the dimension that was summed over).

If the final rank is 1, we reshape it to `(batch\_size, 1)`.

# Examples

Assume `x = [[1, 2], [3, 4]]` and `y = [[5, 6], [7, 8]]`

`batch\_dot(x, y, axes=1) = [[17], [53]]` which is the main diagonal

of `x.dot(y.T)`, although we never have to calculate the off-diagonal

elements.

Pseudocode:

```

inner\_products = []

for xi, yi in zip(x, y):

inner\_products.append(xi.dot(yi))

result = stack(inner\_products)

```

Shape inference:

Let `x`'s shape be `(100, 20)` and `y`'s shape be `(100, 30, 20)`.

If `axes` is (1, 2), to find the output shape of resultant tensor,

loop through each dimension in `x`'s shape and `y`'s shape:

\* `x.shape[0]` : 100 : append to output shape

\* `x.shape[1]` : 20 : do not append to output shape,

dimension 1 of `x` has been summed over. (`dot\_axes[0]` = 1)

\* `y.shape[0]` : 100 : do not append to output shape,

always ignore first dimension of `y`

\* `y.shape[1]` : 30 : append to output shape

\* `y.shape[2]` : 20 : do not append to output shape,

dimension 2 of `y` has been summed over. (`dot\_axes[1]` = 2)

`output\_shape` = `(100, 30)`

```python

>>> x\_batch = K.ones(shape=(32, 20, 1))

>>> y\_batch = K.ones(shape=(32, 30, 20))

>>> xy\_batch\_dot = K.batch\_dot(x\_batch, y\_batch, axes=(1, 2))

>>> K.int\_shape(xy\_batch\_dot)

(32, 1, 30)

```

{{np\_implementation}}

"""

x\_shape = int\_shape(x)

y\_shape = int\_shape(y)

x\_ndim = len(x\_shape)

y\_ndim = len(y\_shape)

if x\_ndim < 2 or y\_ndim < 2:

raise ValueError('Can not do batch\_dot on inputs '

'with rank < 2. '

'Received inputs with shapes ' +

str(x\_shape) + ' and ' +

str(y\_shape) + '.')

x\_batch\_size = x\_shape[0]

y\_batch\_size = y\_shape[0]

if x\_batch\_size is not None and y\_batch\_size is not None:

if x\_batch\_size != y\_batch\_size:

raise ValueError('Can not do batch\_dot on inputs '

'with different batch sizes. '

'Received inputs with shapes ' +

str(x\_shape) + ' and ' +

str(y\_shape) + '.')

if isinstance(axes, int):

axes = [axes, axes]

if axes is None:

if y\_ndim == 2:

axes = [x\_ndim - 1, y\_ndim - 1]

else:

axes = [x\_ndim - 1, y\_ndim - 2]

if py\_any([isinstance(a, (list, tuple)) for a in axes]):

raise ValueError('Multiple target dimensions are not supported. ' +

'Expected: None, int, (int, int), ' +

'Provided: ' + str(axes))

# if tuple, convert to list.

axes = list(axes)

# convert negative indices.

if axes[0] < 0:

axes[0] += x\_ndim

if axes[1] < 0:

axes[1] += y\_ndim

# sanity checks

if 0 in axes:

raise ValueError('Can not perform batch\_dot over axis 0.'

'If your inputs are not batched,'

' add a dummy batch dimension to your '

'inputs using K.expand\_dims(x, 0)')

a0, a1 = axes

d1 = x\_shape[a0]

d2 = y\_shape[a1]

if d1 is not None and d2 is not None and d1 != d2:

raise ValueError('Can not do batch\_dot on inputs with shapes ' +

str(x\_shape) + ' and ' + str(y\_shape) +

' with axes=' + str(axes) + '. x.shape[%d] != '

'y.shape[%d] (%d != %d).' % (axes[0], axes[1], d1, d2))

# backup ndims. Need them later.

orig\_x\_ndim = x\_ndim

orig\_y\_ndim = y\_ndim

# if rank is 2, expand to 3.

if x\_ndim == 2:

x = tf.expand\_dims(x, 1)

a0 += 1

x\_ndim += 1

if y\_ndim == 2:

y = tf.expand\_dims(y, 2)

y\_ndim += 1

# bring x's dimension to be reduced to last axis.

if a0 != x\_ndim - 1:

pattern = list(range(x\_ndim))

for i in range(a0, x\_ndim - 1):

pattern[i] = pattern[i + 1]

pattern[-1] = a0

x = tf.transpose(x, pattern)

# bring y's dimension to be reduced to axis 1.

if a1 != 1:

pattern = list(range(y\_ndim))

for i in range(a1, 1, -1):

pattern[i] = pattern[i - 1]

pattern[1] = a1

y = tf.transpose(y, pattern)

# normalize both inputs to rank 3.

if x\_ndim > 3:

# squash middle dimensions of x.

x\_shape = shape(x)

x\_mid\_dims = x\_shape[1:-1]

x\_squashed\_dim = tf.reduce\_prod(x\_mid\_dims)

x\_squashed\_shape = tf.stack([x\_shape[0], x\_squashed\_dim, x\_shape[-1]])

x = tf.reshape(x, x\_squashed\_shape)

x\_squashed = True

else:

x\_squashed = False

if y\_ndim > 3:

# squash trailing dimensions of y.

y\_shape = shape(y)

y\_trail\_dims = y\_shape[2:]

y\_squashed\_dim = tf.reduce\_prod(y\_trail\_dims)

y\_squashed\_shape = tf.stack([y\_shape[0], y\_shape[1], y\_squashed\_dim])

y = tf.reshape(y, y\_squashed\_shape)

y\_squashed = True

else:

y\_squashed = False

result = tf.matmul(x, y)

# if inputs were squashed, we have to reshape the matmul output.

output\_shape = tf.shape(result)

do\_reshape = False

if x\_squashed:

output\_shape = tf.concat([output\_shape[:1],

x\_mid\_dims,

output\_shape[-1:]], 0)

do\_reshape = True

if y\_squashed:

output\_shape = tf.concat([output\_shape[:-1], y\_trail\_dims], 0)

do\_reshape = True

if do\_reshape:

result = tf.reshape(result, output\_shape)

# if the inputs were originally rank 2, we remove the added 1 dim.

if orig\_x\_ndim == 2:

result = tf.squeeze(result, 1)

elif orig\_y\_ndim == 2:

result = tf.squeeze(result, -1)

return result

def transpose(x):

"""Transposes a tensor and returns it.

# Arguments

x: Tensor or variable.

# Returns

A tensor.

# Examples

```python

>>> var = K.variable([[1, 2, 3], [4, 5, 6]])

>>> K.eval(var)

array([[ 1., 2., 3.],

[ 4., 5., 6.]], dtype=float32)

>>> var\_transposed = K.transpose(var)

>>> K.eval(var\_transposed)

array([[ 1., 4.],

[ 2., 5.],

[ 3., 6.]], dtype=float32)

```

```python

>>> inputs = K.placeholder((2, 3))

>>> inputs

<tf.Tensor 'Placeholder\_11:0' shape=(2, 3) dtype=float32>

>>> input\_transposed = K.transpose(inputs)

>>> input\_transposed

<tf.Tensor 'transpose\_4:0' shape=(3, 2) dtype=float32>

```

{{np\_implementation}}

"""

return tf.transpose(x)

def gather(reference, indices):

"""Retrieves the elements of indices `indices` in the tensor `reference`.

# Arguments

reference: A tensor.

indices: An integer tensor of indices.

# Returns

A tensor of same type as `reference`.

{{np\_implementation}}

"""

return tf.nn.embedding\_lookup(reference, indices)

# ELEMENT-WISE OPERATIONS

def max(x, axis=None, keepdims=False):

"""Maximum value in a tensor.

# Arguments

x: A tensor or variable.

axis: An integer or list of integers in [-rank(x), rank(x)),

the axes to find maximum values. If `None` (default), finds the

maximum over all dimensions.

keepdims: A boolean, whether to keep the dimensions or not.

If `keepdims` is `False`, the rank of the tensor is reduced

by 1. If `keepdims` is `True`,

the reduced dimension is retained with length 1.

# Returns

A tensor with maximum values of `x`.

{{np\_implementation}}

"""

return tf.reduce\_max(x, axis, keepdims)

def min(x, axis=None, keepdims=False):

"""Minimum value in a tensor.

# Arguments

x: A tensor or variable.

axis: An integer or list of integers in [-rank(x), rank(x)),

the axes to find minimum values. If `None` (default), finds the

minimum over all dimensions.

keepdims: A boolean, whether to keep the dimensions or not.

If `keepdims` is `False`, the rank of the tensor is reduced

by 1. If `keepdims` is `True`,

the reduced dimension is retained with length 1.

# Returns

A tensor with miminum values of `x`.

{{np\_implementation}}

"""

return tf.reduce\_min(x, axis, keepdims)

def sum(x, axis=None, keepdims=False):

"""Sum of the values in a tensor, alongside the specified axis.

# Arguments

x: A tensor or variable.

axis: An integer or list of integers in [-rank(x), rank(x)),

the axes to sum over. If `None` (default), sums over all

dimensions.

keepdims: A boolean, whether to keep the dimensions or not.

If `keepdims` is `False`, the rank of the tensor is reduced

by 1. If `keepdims` is `True`,

the reduced dimension is retained with length 1.

# Returns

A tensor with sum of `x`.

{{np\_implementation}}

"""

return tf.reduce\_sum(x, axis, keepdims)

def prod(x, axis=None, keepdims=False):

"""Multiplies the values in a tensor, alongside the specified axis.

# Arguments

x: A tensor or variable.

axis: An integer or list of integers in [-rank(x), rank(x)),

the axes to compute the product. If `None` (default), computes

the product over all dimensions.

keepdims: A boolean, whether to keep the dimensions or not.

If `keepdims` is `False`, the rank of the tensor is reduced

by 1. If `keepdims` is `True`,

the reduced dimension is retained with length 1.

# Returns

A tensor with the product of elements of `x`.

{{np\_implementation}}

"""

return tf.reduce\_prod(x, axis, keepdims)

def cumsum(x, axis=0):

"""Cumulative sum of the values in a tensor, alongside the specified axis.

# Arguments

x: A tensor or variable.

axis: An integer, the axis to compute the sum.

# Returns

A tensor of the cumulative sum of values of `x` along `axis`.

{{np\_implementation}}

"""

return tf\_math\_ops.cumsum(x, axis=axis)

def cumprod(x, axis=0):

"""Cumulative product of the values in a tensor, alongside the specified axis.

# Arguments

x: A tensor or variable.

axis: An integer, the axis to compute the product.

# Returns

A tensor of the cumulative product of values of `x` along `axis`.

{{np\_implementation}}

"""

return tf\_math\_ops.cumprod(x, axis=axis)

def var(x, axis=None, keepdims=False):

"""Variance of a tensor, alongside the specified axis.

# Arguments

x: A tensor or variable.

axis: An integer or list of integers in [-rank(x), rank(x)),

the axes to compute the variance. If `None` (default), computes

the variance over all dimensions.

keepdims: A boolean, whether to keep the dimensions or not.

If `keepdims` is `False`, the rank of the tensor is reduced

by 1. If `keepdims` is `True`,

the reduced dimension is retained with length 1.

# Returns

A tensor with the variance of elements of `x`.

{{np\_implementation}}

"""

if x.dtype.base\_dtype == tf.bool:

x = tf.cast(x, floatx())

m = tf.reduce\_mean(x, axis, True)

devs\_squared = tf.square(x - m)

return tf.reduce\_mean(devs\_squared,

axis,

keepdims)

def std(x, axis=None, keepdims=False):

"""Standard deviation of a tensor, alongside the specified axis.

# Arguments

x: A tensor or variable.

axis: An integer or list of integers in [-rank(x), rank(x)),

the axes to compute the standard deviation. If `None` (default),

computes the standard deviation over all dimensions.

keepdims: A boolean, whether to keep the dimensions or not.

If `keepdims` is `False`, the rank of the tensor is reduced

by 1. If `keepdims` is `True`,

the reduced dimension is retained with length 1.

# Returns

A tensor with the standard deviation of elements of `x`.

{{np\_implementation}}

"""

return tf.sqrt(var(x, axis=axis, keepdims=keepdims))

def mean(x, axis=None, keepdims=False):

"""Mean of a tensor, alongside the specified axis.

# Arguments

x: A tensor or variable.

axis: An integer or list of integers in [-rank(x), rank(x)),

the axes to compute the mean. If `None` (default), computes

the mean over all dimensions.

keepdims: A boolean, whether to keep the dimensions or not.

If `keepdims` is `False`, the rank of the tensor is reduced

by 1 for each entry in `axis`. If `keepdims` is `True`,

the reduced dimensions are retained with length 1.

# Returns

A tensor with the mean of elements of `x`.

{{np\_implementation}}

"""

if x.dtype.base\_dtype == tf.bool:

x = tf.cast(x, floatx())

return tf.reduce\_mean(x, axis, keepdims)

def any(x, axis=None, keepdims=False):

"""Bitwise reduction (logical OR).

# Arguments

x: Tensor or variable.

axis: An integer or list of integers in [-rank(x), rank(x)),

the axes to compute the logical or. If `None` (default), computes

the logical or over all dimensions.

keepdims: whether the drop or broadcast the reduction axes.

# Returns

A uint8 tensor (0s and 1s).

{{np\_implementation}}

"""

x = tf.cast(x, tf.bool)

return tf.reduce\_any(x, axis, keepdims)

def all(x, axis=None, keepdims=False):

"""Bitwise reduction (logical AND).

# Arguments

x: Tensor or variable.

axis: An integer or list of integers in [-rank(x), rank(x)),

the axes to compute the logical and. If `None` (default), computes

the logical and over all dimensions.

keepdims: whether the drop or broadcast the reduction axes.

# Returns

A uint8 tensor (0s and 1s).

{{np\_implementation}}

"""

x = tf.cast(x, tf.bool)

return tf.reduce\_all(x, axis, keepdims)

def argmax(x, axis=-1):

"""Returns the index of the maximum value along an axis.

# Arguments

x: Tensor or variable.

axis: axis along which to perform the reduction.

# Returns

A tensor.

{{np\_implementation}}

"""

return tf.argmax(x, axis)

def argmin(x, axis=-1):

"""Returns the index of the minimum value along an axis.

# Arguments

x: Tensor or variable.

axis: axis along which to perform the reduction.

# Returns

A tensor.

{{np\_implementation}}

"""

return tf.argmin(x, axis)

def square(x):

"""Element-wise square.

# Arguments

x: Tensor or variable.

# Returns

A tensor.

"""

return tf.square(x)

def abs(x):

"""Element-wise absolute value.

# Arguments

x: Tensor or variable.

# Returns

A tensor.

"""

return tf.abs(x)

def sqrt(x):

"""Element-wise square root.

# Arguments

x: Tensor or variable.

# Returns

A tensor.

{{np\_implementation}}

"""

zero = \_to\_tensor(0., x.dtype.base\_dtype)

inf = \_to\_tensor(np.inf, x.dtype.base\_dtype)

x = tf.clip\_by\_value(x, zero, inf)

return tf.sqrt(x)

def exp(x):

"""Element-wise exponential.

# Arguments

x: Tensor or variable.

# Returns

A tensor.

"""

return tf.exp(x)

def log(x):

"""Element-wise log.

# Arguments

x: Tensor or variable.

# Returns

A tensor.

"""

return tf\_math\_ops.log(x)

def logsumexp(x, axis=None, keepdims=False):

"""Computes log(sum(exp(elements across dimensions of a tensor))).

This function is more numerically stable than log(sum(exp(x))).

It avoids overflows caused by taking the exp of large inputs and

underflows caused by taking the log of small inputs.

# Arguments

x: A tensor or variable.

axis: axis: An integer or list of integers in [-rank(x), rank(x)),

the axes to compute the logsumexp. If `None` (default), computes

the logsumexp over all dimensions.

keepdims: A boolean, whether to keep the dimensions or not.

If `keepdims` is `False`, the rank of the tensor is reduced

by 1. If `keepdims` is `True`, the reduced dimension is

retained with length 1.

# Returns

The reduced tensor.

{{np\_implementation}}

"""

return tf.reduce\_logsumexp(x, axis, keepdims)

def round(x):

"""Element-wise rounding to the closest integer.

In case of tie, the rounding mode used is "half to even".

# Arguments

x: Tensor or variable.

# Returns

A tensor.

"""

return tf.round(x)

def sign(x):

"""Element-wise sign.

# Arguments

x: Tensor or variable.

# Returns

A tensor.

"""

return tf.sign(x)

def pow(x, a):

"""Element-wise exponentiation.

# Arguments

x: Tensor or variable.

a: Python integer.

# Returns

A tensor.

{{np\_implementation}}

"""

return tf.pow(x, a)

def clip(x, min\_value, max\_value):

"""Element-wise value clipping.

# Arguments

x: Tensor or variable.

min\_value: Python float, integer or tensor.

max\_value: Python float, integer or tensor.

# Returns

A tensor.

{{np\_implementation}}

"""

if (isinstance(min\_value, (int, float)) and

isinstance(max\_value, (int, float))):

if max\_value < min\_value:

max\_value = min\_value

if min\_value is None:

min\_value = -np.inf

if max\_value is None:

max\_value = np.inf

return tf.clip\_by\_value(x, min\_value, max\_value)

def equal(x, y):

"""Element-wise equality between two tensors.

# Arguments

x: Tensor or variable.

y: Tensor or variable.

# Returns

A bool tensor.

{{np\_implementation}}

"""

return tf.equal(x, y)

def not\_equal(x, y):

"""Element-wise inequality between two tensors.

# Arguments

x: Tensor or variable.

y: Tensor or variable.

# Returns

A bool tensor.

{{np\_implementation}}

"""

return tf.not\_equal(x, y)

def greater(x, y):

"""Element-wise truth value of (x > y).

# Arguments

x: Tensor or variable.

y: Tensor or variable.

# Returns

A bool tensor.

{{np\_implementation}}

"""

return tf.greater(x, y)

def greater\_equal(x, y):

"""Element-wise truth value of (x >= y).

# Arguments

x: Tensor or variable.

y: Tensor or variable.

# Returns

A bool tensor.

{{np\_implementation}}

"""

return tf.greater\_equal(x, y)

def less(x, y):

"""Element-wise truth value of (x < y).

# Arguments

x: Tensor or variable.

y: Tensor or variable.

# Returns

A bool tensor.

{{np\_implementation}}

"""

return tf.less(x, y)

def less\_equal(x, y):

"""Element-wise truth value of (x <= y).

# Arguments

x: Tensor or variable.

y: Tensor or variable.

# Returns

A bool tensor.

{{np\_implementation}}

"""

return tf.less\_equal(x, y)

def maximum(x, y):

"""Element-wise maximum of two tensors.

# Arguments

x: Tensor or variable.

y: Tensor or variable.

# Returns

A tensor.

{{np\_implementation}}

"""

return tf.maximum(x, y)

def minimum(x, y):

"""Element-wise minimum of two tensors.

# Arguments

x: Tensor or variable.

y: Tensor or variable.

# Returns

A tensor.

{{np\_implementation}}

"""

return tf.minimum(x, y)

def sin(x):

"""Computes sin of x element-wise.

# Arguments

x: Tensor or variable.

# Returns

A tensor.

"""

return tf.sin(x)

def cos(x):

"""Computes cos of x element-wise.

# Arguments

x: Tensor or variable.

# Returns

A tensor.

"""

return tf.cos(x)

def \_regular\_normalize\_batch\_in\_training(x, gamma, beta,

reduction\_axes, epsilon=1e-3):

"""Non-fused version of `normalize\_batch\_in\_training`.

# Arguments

x: Input tensor or variable.

gamma: Tensor by which to scale the input.

beta: Tensor with which to center the input.

reduction\_axes: iterable of integers,

axes over which to normalize.

epsilon: Fuzz factor.

# Returns

A tuple length of 3, `(normalized\_tensor, mean, variance)`.

"""

mean, var = tf.nn.moments(x, reduction\_axes,

None, None, False)

normed = tf.nn.batch\_normalization(x, mean, var,

beta, gamma,

epsilon)

return normed, mean, var

def \_broadcast\_normalize\_batch\_in\_training(x, gamma, beta,

reduction\_axes, epsilon=1e-3):

"""Non-fused, broadcast version of `normalize\_batch\_in\_training`.

# Arguments

x: Input tensor or variable.

gamma: Tensor by which to scale the input.

beta: Tensor with which to center the input.

reduction\_axes: iterable of integers,

axes over which to normalize.

epsilon: Fuzz factor.

# Returns

A tuple length of 3, `(normalized\_tensor, mean, variance)`.

"""

mean, var = tf.nn.moments(x, reduction\_axes,

None, None, False)

target\_shape = []

for axis in range(ndim(x)):

if axis in reduction\_axes:

target\_shape.append(1)

else:

target\_shape.append(tf.shape(x)[axis])

target\_shape = tf.stack(target\_shape)

broadcast\_mean = tf.reshape(mean, target\_shape)

broadcast\_var = tf.reshape(var, target\_shape)

if gamma is None:

broadcast\_gamma = None

else:

broadcast\_gamma = tf.reshape(gamma, target\_shape)

if beta is None:

broadcast\_beta = None

else:

broadcast\_beta = tf.reshape(beta, target\_shape)

normed = tf.nn.batch\_normalization(

x,

broadcast\_mean,

broadcast\_var,

broadcast\_beta,

broadcast\_gamma,

epsilon)

return normed, mean, var

def \_fused\_normalize\_batch\_in\_training(x, gamma, beta, reduction\_axes,

epsilon=1e-3):

"""Fused version of `normalize\_batch\_in\_training`.

# Arguments

x: Input tensor or variable.

gamma: Tensor by which to scale the input.

beta: Tensor with which to center the input.

reduction\_axes: iterable of integers,

axes over which to normalize.

epsilon: Fuzz factor.

# Returns

A tuple length of 3, `(normalized\_tensor, mean, variance)`.

"""

if list(reduction\_axes) == [0, 1, 2]:

normalization\_axis = 3

tf\_data\_format = 'NHWC'

else:

normalization\_axis = 1

tf\_data\_format = 'NCHW'

if gamma is None:

gamma = tf.constant(1.0,

dtype=x.dtype,

shape=[x.shape[normalization\_axis]])

if beta is None:

beta = tf.constant(0.0,

dtype=x.dtype,

shape=[x.shape[normalization\_axis]])

if gamma.dtype != tf.float32:

gamma = tf.cast(gamma, tf.float32)

if beta.dtype != tf.float32:

beta = tf.cast(beta, tf.float32)

if \_has\_compat\_v1:

fused\_batch\_norm = tf.compat.v1.nn.fused\_batch\_norm

else:

fused\_batch\_norm = tf.nn.fused\_batch\_norm

return fused\_batch\_norm(

x,

gamma,

beta,

epsilon=epsilon,

data\_format=tf\_data\_format)

def normalize\_batch\_in\_training(x, gamma, beta,

reduction\_axes, epsilon=1e-3):

"""Computes mean and std for batch then apply batch\_normalization on batch.

# Arguments

x: Input tensor or variable.

gamma: Tensor by which to scale the input.

beta: Tensor with which to center the input.

reduction\_axes: iterable of integers,

axes over which to normalize.

epsilon: Fuzz factor.

# Returns

A tuple length of 3, `(normalized\_tensor, mean, variance)`.

"""

if (ndim(x) == 4 and

list(reduction\_axes) in [[0, 1, 2], [0, 2, 3]] and

\_is\_tf\_1()):

if not \_has\_nchw\_support() and list(reduction\_axes) == [0, 2, 3]:

return \_broadcast\_normalize\_batch\_in\_training(x, gamma, beta,

reduction\_axes,

epsilon=epsilon)

return \_fused\_normalize\_batch\_in\_training(

x, gamma, beta, reduction\_axes,

epsilon=epsilon)

else:

if sorted(reduction\_axes) == list(range(ndim(x)))[:-1]:

return \_regular\_normalize\_batch\_in\_training(x, gamma, beta,

reduction\_axes,

epsilon=epsilon)

else:

return \_broadcast\_normalize\_batch\_in\_training(x, gamma, beta,

reduction\_axes,

epsilon=epsilon)

def batch\_normalization(x, mean, var, beta, gamma, axis=-1, epsilon=1e-3):

"""Applies batch normalization on x given mean, var, beta and gamma.

I.e. returns:

`output = (x - mean) / sqrt(var + epsilon) \* gamma + beta`

# Arguments

x: Input tensor or variable.

mean: Mean of batch.

var: Variance of batch.

beta: Tensor with which to center the input.

gamma: Tensor by which to scale the input.

axis: Integer, the axis that should be normalized.

(typically the features axis).

epsilon: Fuzz factor.

# Returns

A tensor.

{{np\_implementation}}

"""

if ndim(x) == 4:

# The CPU implementation of FusedBatchNorm only support NHWC

if axis == 1 or axis == -3:

tf\_data\_format = 'NCHW'

elif axis == 3 or axis == -1:

tf\_data\_format = 'NHWC'

else:

tf\_data\_format = None

if ((tf\_data\_format == 'NHWC' or

(tf\_data\_format == 'NCHW' and

\_has\_nchw\_support())) and

\_is\_tf\_1()):

# The mean / var / beta / gamma may be processed by broadcast

# so it may have extra axes with 1,

# it is not needed and should be removed

if ndim(mean) > 1:

mean = tf.reshape(mean, [-1])

if ndim(var) > 1:

var = tf.reshape(var, [-1])

if beta is None:

beta = zeros\_like(mean)

elif ndim(beta) > 1:

beta = tf.reshape(beta, [-1])

if gamma is None:

gamma = ones\_like(mean)

elif ndim(gamma) > 1:

gamma = tf.reshape(gamma, [-1])

if gamma.dtype != tf.float32:

gamma = tf.cast(gamma, tf.float32)

if beta.dtype != tf.float32:

beta = tf.cast(beta, tf.float32)

if mean.dtype != tf.float32:

mean = tf.cast(mean, tf.float32)

if var.dtype != tf.float32:

var = tf.cast(var, tf.float32)

if \_has\_compat\_v1:

fused\_batch\_norm = tf.compat.v1.nn.fused\_batch\_norm

else:

fused\_batch\_norm = tf.nn.fused\_batch\_norm

y, \_, \_ = fused\_batch\_norm(

x,

gamma,

beta,

epsilon=epsilon,

mean=mean,

variance=var,

data\_format=tf\_data\_format,

is\_training=False

)

return y

# default

return tf.nn.batch\_normalization(x, mean, var, beta, gamma, epsilon)

# SHAPE OPERATIONS

def concatenate(tensors, axis=-1):

"""Concatenates a list of tensors alongside the specified axis.

# Arguments

tensors: list of tensors to concatenate.

axis: concatenation axis.

# Returns

A tensor.

"""

if axis < 0:

rank = ndim(tensors[0])

if rank:

axis %= rank

else:

axis = 0

if py\_all([is\_sparse(x) for x in tensors]):

return tf.sparse.concat(axis, tensors)

else:

return tf.concat([to\_dense(x) for x in tensors], axis)

def reshape(x, shape):

"""Reshapes a tensor to the specified shape.

# Arguments

x: Tensor or variable.

shape: Target shape tuple.

# Returns

A tensor.

"""

return tf.reshape(x, shape)

def permute\_dimensions(x, pattern):

"""Permutes axes in a tensor.

# Arguments

x: Tensor or variable.

pattern: A tuple of

dimension indices, e.g. `(0, 2, 1)`.

# Returns

A tensor.

"""

return tf.transpose(x, perm=pattern)

def resize\_images(x,

height\_factor,

width\_factor,

data\_format,

interpolation='nearest'):

"""Resizes the images contained in a 4D tensor.

# Arguments

x: Tensor or variable to resize.

height\_factor: Positive integer.

width\_factor: Positive integer.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

interpolation: A string, one of `nearest` or `bilinear`.

# Returns

A tensor.

# Raises

ValueError: if `data\_format` is

neither `"channels\_last"` or `"channels\_first"`.

"""

if data\_format == 'channels\_first':

rows, cols = 2, 3

else:

rows, cols = 1, 2

original\_shape = int\_shape(x)

new\_shape = tf.shape(x)[rows:cols + 1]

new\_shape \*= tf.constant(np.array([height\_factor, width\_factor],

dtype='int32'))

if data\_format == 'channels\_first':

x = permute\_dimensions(x, [0, 2, 3, 1])

if interpolation == 'nearest':

x = tf\_image\_ops.resize\_nearest\_neighbor(x, new\_shape)

elif interpolation == 'bilinear':

x = tf\_image\_ops.resize\_bilinear(x, new\_shape)

else:

raise ValueError('interpolation should be one '

'of "nearest" or "bilinear".')

if data\_format == 'channels\_first':

x = permute\_dimensions(x, [0, 3, 1, 2])

if original\_shape[rows] is None:

new\_height = None

else:

new\_height = original\_shape[rows] \* height\_factor

if original\_shape[cols] is None:

new\_width = None

else:

new\_width = original\_shape[cols] \* width\_factor

output\_shape = (None, new\_height, new\_width, None)

x.set\_shape(transpose\_shape(output\_shape, data\_format,

spatial\_axes=(1, 2)))

return x

def resize\_volumes(x, depth\_factor, height\_factor, width\_factor, data\_format):

"""Resizes the volume contained in a 5D tensor.

# Arguments

x: Tensor or variable to resize.

depth\_factor: Positive integer.

height\_factor: Positive integer.

width\_factor: Positive integer.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

# Returns

A tensor.

# Raises

ValueError: if `data\_format` is

neither `"channels\_last"` or `"channels\_first"`.

"""

if data\_format == 'channels\_first':

output = repeat\_elements(x, depth\_factor, axis=2)

output = repeat\_elements(output, height\_factor, axis=3)

output = repeat\_elements(output, width\_factor, axis=4)

return output

elif data\_format == 'channels\_last':

output = repeat\_elements(x, depth\_factor, axis=1)

output = repeat\_elements(output, height\_factor, axis=2)

output = repeat\_elements(output, width\_factor, axis=3)

return output

else:

raise ValueError('Unknown data\_format: ' + str(data\_format))

def repeat\_elements(x, rep, axis):

"""Repeats the elements of a tensor along an axis, like `np.repeat`.

If `x` has shape `(s1, s2, s3)` and `axis` is `1`, the output

will have shape `(s1, s2 \* rep, s3)`.

# Arguments

x: Tensor or variable.

rep: Python integer, number of times to repeat.

axis: Axis along which to repeat.

# Returns

A tensor.

"""

x\_shape = x.shape.as\_list()

# For static axis

if x\_shape[axis] is not None:

# slices along the repeat axis

splits = tf.split(value=x, num\_or\_size\_splits=x\_shape[axis], axis=axis)

# repeat each slice the given number of reps

x\_rep = [s for s in splits for \_ in range(rep)]

return concatenate(x\_rep, axis)

# Here we use tf.tile to mimic behavior of np.repeat so that

# we can handle dynamic shapes (that include None).

# To do that, we need an auxiliary axis to repeat elements along

# it and then merge them along the desired axis.

# Repeating

auxiliary\_axis = axis + 1

x\_shape = tf.shape(x)

x\_rep = tf.expand\_dims(x, axis=auxiliary\_axis)

reps = np.ones(len(x.shape) + 1)

reps[auxiliary\_axis] = rep

x\_rep = tf.tile(x\_rep, reps)

# Merging

reps = np.delete(reps, auxiliary\_axis)

reps[axis] = rep

reps = tf.constant(reps, dtype='int32')

x\_shape = x\_shape \* reps

x\_rep = tf.reshape(x\_rep, x\_shape)

# Fix shape representation

x\_shape = x.shape.as\_list()

x\_rep.set\_shape(x\_shape)

x\_rep.\_keras\_shape = tuple(x\_shape)

return x\_rep

def repeat(x, n):

"""Repeats a 2D tensor.

if `x` has shape (samples, dim) and `n` is `2`,

the output will have shape `(samples, 2, dim)`.

# Arguments

x: Tensor or variable.

n: Python integer, number of times to repeat.

# Returns

A tensor.

"""

assert ndim(x) == 2

x = tf.expand\_dims(x, 1)

pattern = tf.stack([1, n, 1])

return tf.tile(x, pattern)

def arange(start, stop=None, step=1, dtype='int32'):

"""Creates a 1D tensor containing a sequence of integers.

The function arguments use the same convention as

Theano's arange: if only one argument is provided,

it is in fact the "stop" argument and "start" is 0.

The default type of the returned tensor is `'int32'` to

match TensorFlow's default.

# Arguments

start: Start value.

stop: Stop value.

step: Difference between two successive values.

dtype: Integer dtype to use.

# Returns

An integer tensor.

"""

# Match the behavior of numpy and Theano by returning an empty sequence.

if stop is None:

try:

if start < 0:

start = 0

except TypeError:

# Handle case where start is a tensor

start = tf.cond(start < 0,

true\_fn=lambda: tf.constant(0, dtype=start.dtype),

false\_fn=lambda: start)

result = tf.range(start, limit=stop, delta=step, name='arange')

if dtype != 'int32':

result = cast(result, dtype)

return result

def tile(x, n):

"""Creates a tensor by tiling `x` by `n`.

# Arguments

x: A tensor or variable

n: A list of integer. The length must be the same as the number of

dimensions in `x`.

# Returns

A tiled tensor.

# Example

```python

>>> from keras import backend as K

>>> kvar = K.variable(np.random.random((2, 3)))

>>> kvar\_tile = K.tile(K.eye(2), (2, 3))

>>> K.eval(kvar\_tile)

array([[1., 0., 1., 0., 1., 0.],

[0., 1., 0., 1., 0., 1.],

[1., 0., 1., 0., 1., 0.],

[0., 1., 0., 1., 0., 1.]], dtype=float32)

```

{{np\_implementation}}

"""

if isinstance(n, int):

n = (n,)

elif isinstance(n, list):

n = tuple(n)

shape = int\_shape(x)

if not is\_tensor(n):

if len(n) < len(shape): # Padding the axis

n = tuple([1 for \_ in range(len(shape) - len(n))]) + n

elif len(n) != len(shape):

raise NotImplementedError

return tf.tile(x, n)

def flatten(x):

"""Flatten a tensor.

# Arguments

x: A tensor or variable.

# Returns

A tensor, reshaped into 1-D

"""

return tf.reshape(x, [-1])

def batch\_flatten(x):

"""Turn a nD tensor into a 2D tensor with same 0th dimension.

In other words, it flattens each data samples of a batch.

# Arguments

x: A tensor or variable.

# Returns

A tensor.

"""

x = tf.reshape(

x, tf.stack([-1, prod(shape(x)[1:])],

name='stack\_' + str(np.random.randint(1e4))))

return x

def expand\_dims(x, axis=-1):

"""Adds a 1-sized dimension at index "axis".

# Arguments

x: A tensor or variable.

axis: Position where to add a new axis.

# Returns

A tensor with expanded dimensions.

"""

return tf.expand\_dims(x, axis)

def squeeze(x, axis):

"""Removes a 1-dimension from the tensor at index "axis".

# Arguments

x: A tensor or variable.

axis: Axis to drop.

# Returns

A tensor with the same data as `x` but reduced dimensions.

"""

return tf.squeeze(x, [axis])

def temporal\_padding(x, padding=(1, 1)):

"""Pads the middle dimension of a 3D tensor.

# Arguments

x: Tensor or variable.

padding: Tuple of 2 integers, how many zeros to

add at the start and end of dim 1.

# Returns

A padded 3D tensor.

"""

assert len(padding) == 2

pattern = [[0, 0], [padding[0], padding[1]], [0, 0]]

return tf.pad(x, pattern)

def spatial\_2d\_padding(x, padding=((1, 1), (1, 1)), data\_format=None):

"""Pads the 2nd and 3rd dimensions of a 4D tensor.

# Arguments

x: Tensor or variable.

padding: Tuple of 2 tuples, padding pattern.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

# Returns

A padded 4D tensor.

# Raises

ValueError: if `data\_format` is

neither `"channels\_last"` or `"channels\_first"`.

"""

assert len(padding) == 2

assert len(padding[0]) == 2

assert len(padding[1]) == 2

data\_format = normalize\_data\_format(data\_format)

pattern = [[0, 0],

list(padding[0]),

list(padding[1]),

[0, 0]]

pattern = transpose\_shape(pattern, data\_format, spatial\_axes=(1, 2))

return tf.pad(x, pattern)

def spatial\_3d\_padding(x, padding=((1, 1), (1, 1), (1, 1)), data\_format=None):

"""Pads 5D tensor with zeros along the depth, height, width dimensions.

Pads these dimensions with respectively

"padding[0]", "padding[1]" and "padding[2]" zeros left and right.

For 'channels\_last' data\_format,

the 2nd, 3rd and 4th dimension will be padded.

For 'channels\_first' data\_format,

the 3rd, 4th and 5th dimension will be padded.

# Arguments

x: Tensor or variable.

padding: Tuple of 3 tuples, padding pattern.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

# Returns

A padded 5D tensor.

# Raises

ValueError: if `data\_format` is

neither `"channels\_last"` or `"channels\_first"`.

"""

assert len(padding) == 3

assert len(padding[0]) == 2

assert len(padding[1]) == 2

assert len(padding[2]) == 2

data\_format = normalize\_data\_format(data\_format)

pattern = [

[0, 0],

[padding[0][0], padding[0][1]],

[padding[1][0], padding[1][1]],

[padding[2][0], padding[2][1]],

[0, 0]

]

pattern = transpose\_shape(pattern, data\_format, spatial\_axes=(1, 2, 3))

return tf.pad(x, pattern)

def stack(x, axis=0):

"""Stacks a list of rank `R` tensors into a rank `R+1` tensor.

# Arguments

x: List of tensors.

axis: Axis along which to perform stacking.

# Returns

A tensor.

{{np\_implementation}}

"""

return tf.stack(x, axis=axis)

def one\_hot(indices, num\_classes):

"""Computes the one-hot representation of an integer tensor.

# Arguments

indices: nD integer tensor of shape

`(batch\_size, dim1, dim2, ... dim(n-1))`

num\_classes: Integer, number of classes to consider.

# Returns

(n + 1)D one hot representation of the input

with shape `(batch\_size, dim1, dim2, ... dim(n-1), num\_classes)`

"""

return tf.one\_hot(indices, depth=num\_classes, axis=-1)

def reverse(x, axes):

"""Reverses a tensor along the specified axes.

# Arguments

x: Tensor to reverse.

axes: Integer or iterable of integers.

Axes to reverse.

# Returns

A tensor.

{{np\_implementation}}

"""

if isinstance(axes, int):

axes = [axes]

return tf.reverse(x, axes)

def slice(x, start, size):

"""Extracts a slice from a tensor.

# Arguments

x: Input tensor.

start: Integer list/tuple or tensor

indicating the start indices of the slice

along each axis.

size: Integer list/tuple or tensor

indicating how many dimensions to slice

along each axis.

# Returns

A sliced tensor:

```python

new\_x = x[start[0]: start[0] + size[0], ..., start[-1]: start[-1] + size[-1]]

```

# Raises

ValueError: if the dimension and the size of indices mismatches.

{{np\_implementation}}

"""

x\_shape = int\_shape(x)

if (x\_shape is not None) and (x\_shape[0] is not None):

len\_start = int\_shape(start)[0] if is\_tensor(start) else len(start)

len\_size = int\_shape(size)[0] if is\_tensor(size) else len(size)

if not (len(int\_shape(x)) == len\_start == len\_size):

raise ValueError('The dimension and the size of indices should match.')

return tf.slice(x, start, size)

# VALUE MANIPULATION

def get\_value(x):

"""Returns the value of a variable.

# Arguments

x: input variable.

# Returns

A Numpy array.

"""

if \_is\_tf\_1():

return x.eval(session=get\_session())

else:

return x.numpy()

def batch\_get\_value(ops):

"""Returns the value of more than one tensor variable.

# Arguments

ops: list of ops to run.

# Returns

A list of Numpy arrays.

"""

return tf\_keras\_backend.batch\_get\_value(ops)

def set\_value(x, value):

"""Sets the value of a variable, from a Numpy array.

# Arguments

x: Variable to set to a new value.

value: Value to set the tensor to, as a Numpy array

(of the same shape).

"""

tf\_keras\_backend.set\_value(x, value)

def batch\_set\_value(tuples):

"""Sets the values of many tensor variables at once.

# Arguments

tuples: a list of tuples `(tensor, value)`.

`value` should be a Numpy array.

"""

tf\_keras\_backend.batch\_set\_value(tuples)

def get\_variable\_shape(x):

"""Returns the shape of a variable.

# Arguments

x: A variable.

# Returns

A tuple of integers.

"""

return int\_shape(x)

@symbolic

def print\_tensor(x, message=''):

"""Prints `message` and the tensor value when evaluated.

Note that `print\_tensor` returns a new tensor identical to `x`

which should be used in the following code. Otherwise the

print operation is not taken into account during evaluation.

# Example

```python

>>> x = K.print\_tensor(x, message="x is: ")

```

# Arguments

x: Tensor to print.

message: Message to print jointly with the tensor.

# Returns

The same tensor `x`, unchanged.

"""

op = tf.print(message, x, output\_stream=sys.stdout)

with tf.control\_dependencies([op]):

return tf.identity(x)

# GRAPH MANIPULATION

def function(inputs, outputs, updates=None, \*\*kwargs):

if \_is\_tf\_1():

v1\_variable\_initialization()

return tf\_keras\_backend.function(inputs, outputs,

updates=updates,

\*\*kwargs)

@symbolic

def gradients(loss, variables):

"""Returns the gradients of `loss` w.r.t. `variables`.

# Arguments

loss: Scalar tensor to minimize.

variables: List of variables.

# Returns

A gradients tensor.

"""

if \_is\_tf\_1():

return tf.gradients(loss, variables, colocate\_gradients\_with\_ops=True)

return tf.gradients(loss, variables)

@symbolic

def stop\_gradient(variables):

"""Returns `variables` but with zero gradient w.r.t. every other variable.

# Arguments

variables: tensor or list of tensors to consider constant with respect

to any other variable.

# Returns

A single tensor or a list of tensors (depending on the passed argument)

that has constant gradient with respect to any other variable.

"""

if isinstance(variables, (list, tuple)):

return map(tf.stop\_gradient, variables)

else:

return tf.stop\_gradient(variables)

# CONTROL FLOW

def rnn(step\_function, inputs, initial\_states,

go\_backwards=False, mask=None, constants=None,

unroll=False, input\_length=None):

"""Iterates over the time dimension of a tensor.

# Arguments

step\_function:

Parameters:

inputs: Tensor with shape (samples, ...) (no time dimension),

representing input for the batch of samples at a certain

time step.

states: List of tensors.

Returns:

outputs: Tensor with shape (samples, ...) (no time dimension),

new\_states: List of tensors, same length and shapes

as 'states'.

inputs: Tensor of temporal data of shape (samples, time, ...)

(at least 3D).

initial\_states: Tensor with shape (samples, ...) (no time dimension),

containing the initial values for the states used in

the step function.

go\_backwards: Boolean. If True, do the iteration over the time

dimension in reverse order and return the reversed sequence.

mask: Binary tensor with shape (samples, time),

with a zero for every element that is masked.

constants: A list of constant values passed at each step.

unroll: Whether to unroll the RNN or to use a symbolic loop

(`while\_loop` or `scan` depending on backend).

input\_length: Static number of timesteps in the input.

# Returns

A tuple, `(last\_output, outputs, new\_states)`.

last\_output: The latest output of the rnn, of shape `(samples, ...)`

outputs: Tensor with shape `(samples, time, ...)` where each

entry `outputs[s, t]` is the output of the step function

at time `t` for sample `s`.

new\_states: List of tensors, latest states returned by

the step function, of shape `(samples, ...)`.

# Raises

ValueError: If input dimension is less than 3.

ValueError: If `unroll` is `True`

but input timestep is not a fixed number.

ValueError: If `mask` is provided (not `None`)

but states is not provided (`len(states)` == 0).

{{np\_implementation}}

"""

last\_output, outputs, new\_states = tf\_keras\_backend.rnn(

step\_function, inputs, initial\_states,

go\_backwards=go\_backwards,

mask=mask,

constants=constants,

unroll=unroll,

input\_length=input\_length)

reachable = tf\_utils.get\_reachable\_from\_inputs([learning\_phase()],

targets=[last\_output])

if last\_output in reachable:

last\_output.\_uses\_learning\_phase = True

return last\_output, outputs, new\_states

@symbolic

def switch(condition, then\_expression, else\_expression):

"""Switches between two operations depending on a scalar value.

Note that both `then\_expression` and `else\_expression`

should be symbolic tensors of the \*same shape\*.

# Arguments

condition: tensor (`int` or `bool`).

then\_expression: either a tensor, or a callable that returns a tensor.

else\_expression: either a tensor, or a callable that returns a tensor.

# Returns

The selected tensor.

# Raises

ValueError: If rank of `condition` is greater than rank of expressions.

{{np\_implementation}}

"""

if condition.dtype != tf.bool:

condition = tf.cast(condition, 'bool')

cond\_ndim = ndim(condition)

if not cond\_ndim:

if not callable(then\_expression):

def then\_expression\_fn():

return then\_expression

else:

then\_expression\_fn = then\_expression

if not callable(else\_expression):

def else\_expression\_fn():

return else\_expression

else:

else\_expression\_fn = else\_expression

x = tf.cond(condition,

then\_expression\_fn,

else\_expression\_fn)

else:

# tf.where needs its condition tensor

# to be the same shape as its two

# result tensors

if callable(then\_expression):

then\_expression = then\_expression()

if callable(else\_expression):

else\_expression = else\_expression()

expr\_ndim = ndim(then\_expression)

if cond\_ndim > expr\_ndim:

raise ValueError('Rank of `condition` should be less than or'

' equal to rank of `then\_expression` and '

'`else\_expression`. ndim(condition)=' +

str(cond\_ndim) + ', ndim(then\_expression)'

'=' + str(expr\_ndim))

if cond\_ndim > 1:

ndim\_diff = expr\_ndim - cond\_ndim

cond\_shape = tf.concat([tf.shape(condition), [1] \* ndim\_diff], axis=0)

condition = tf.reshape(condition, cond\_shape)

expr\_shape = tf.shape(then\_expression)

shape\_diff = expr\_shape - cond\_shape

zero\_expr\_shape = tf.ones\_like(expr\_shape)

tile\_shape = tf.where(shape\_diff > 0, expr\_shape, zero\_expr\_shape)

condition = tf.tile(condition, tile\_shape)

x = tf.where(condition, then\_expression, else\_expression)

return x

@symbolic

def in\_train\_phase(x, alt, training=None):

"""Selects `x` in train phase, and `alt` otherwise.

Note that `alt` should have the \*same shape\* as `x`.

# Arguments

x: What to return in train phase

(tensor or callable that returns a tensor).

alt: What to return otherwise

(tensor or callable that returns a tensor).

training: Optional scalar tensor

(or Python boolean, or Python integer)

specifying the learning phase.

# Returns

Either `x` or `alt` based on the `training` flag.

the `training` flag defaults to `K.learning\_phase()`.

"""

if training is None:

training = learning\_phase()

uses\_learning\_phase = True

else:

uses\_learning\_phase = False

if training is 1 or training is True:

if callable(x):

return x()

else:

return x

elif training is 0 or training is False:

if callable(alt):

return alt()

else:

return alt

# else: assume learning phase is a placeholder tensor.

x = switch(training, x, alt)

if uses\_learning\_phase:

x.\_uses\_learning\_phase = True

return x

@symbolic

def in\_test\_phase(x, alt, training=None):

"""Selects `x` in test phase, and `alt` otherwise.

Note that `alt` should have the \*same shape\* as `x`.

# Arguments

x: What to return in test phase

(tensor or callable that returns a tensor).

alt: What to return otherwise

(tensor or callable that returns a tensor).

training: Optional scalar tensor

(or Python boolean, or Python integer)

specifying the learning phase.

# Returns

Either `x` or `alt` based on `K.learning\_phase`.

"""

return in\_train\_phase(alt, x, training=training)

# NN OPERATIONS

def relu(x, alpha=0., max\_value=None, threshold=0.):

"""Rectified linear unit.

With default values, it returns element-wise `max(x, 0)`.

Otherwise, it follows:

`f(x) = max\_value` for `x >= max\_value`,

`f(x) = x` for `threshold <= x < max\_value`,

`f(x) = alpha \* (x - threshold)` otherwise.

# Arguments

x: A tensor or variable.

alpha: A scalar, slope of negative section (default=`0.`).

max\_value: float. Saturation threshold.

threshold: float. Threshold value for thresholded activation.

# Returns

A tensor.

{{np\_implementation}}

"""

if alpha != 0.:

if max\_value is None and threshold == 0.:

return tf.nn.leaky\_relu(x, alpha=alpha)

if threshold != 0.:

negative\_part = tf.nn.relu(-x + threshold)

else:

negative\_part = tf.nn.relu(-x)

clip\_max = max\_value is not None

if threshold != 0:

# computes x for x > threshold else 0

x = x \* tf.cast(tf.greater(x, threshold), floatx())

elif max\_value == 6:

# if no threshold, then can use nn.relu6 native TF op for performance

x = tf.nn.relu6(x)

clip\_max = False

else:

x = tf.nn.relu(x)

if clip\_max:

max\_value = \_to\_tensor(max\_value, x.dtype.base\_dtype)

zero = \_to\_tensor(0., x.dtype.base\_dtype)

x = tf.clip\_by\_value(x, zero, max\_value)

if alpha != 0:

alpha = \_to\_tensor(alpha, x.dtype.base\_dtype)

x -= alpha \* negative\_part

return x

def elu(x, alpha=1.):

"""Exponential linear unit.

# Arguments

x: A tensor or variable to compute the activation function for.

alpha: A scalar, slope of negative section.

# Returns

A tensor.

{{np\_implementation}}

"""

res = tf.nn.elu(x)

if alpha == 1:

return res

else:

return tf.where(x > 0, res, alpha \* res)

def softmax(x, axis=-1):

"""Softmax of a tensor.

# Arguments

x: A tensor or variable.

axis: The dimension softmax would be performed on.

The default is -1 which indicates the last dimension.

# Returns

A tensor.

{{np\_implementation}}

"""

return tf.nn.softmax(x, axis=axis)

def softplus(x):

"""Softplus of a tensor.

# Arguments

x: A tensor or variable.

# Returns

A tensor.

{{np\_implementation}}

"""

return tf.nn.softplus(x)

def softsign(x):

"""Softsign of a tensor.

# Arguments

x: A tensor or variable.

# Returns

A tensor.

{{np\_implementation}}

"""

return tf.nn.softsign(x)

def categorical\_crossentropy(target, output, from\_logits=False, axis=-1):

"""Categorical crossentropy between an output tensor and a target tensor.

# Arguments

target: A tensor of the same shape as `output`.

output: A tensor resulting from a softmax

(unless `from\_logits` is True, in which

case `output` is expected to be the logits).

from\_logits: Boolean, whether `output` is the

result of a softmax, or is a tensor of logits.

axis: Int specifying the channels axis. `axis=-1`

corresponds to data format `channels\_last`,

and `axis=1` corresponds to data format

`channels\_first`.

# Returns

Output tensor.

# Raises

ValueError: if `axis` is neither -1 nor one of

the axes of `output`.

"""

return tf\_keras\_backend.categorical\_crossentropy(

target, output, from\_logits=from\_logits, axis=axis)

def sparse\_categorical\_crossentropy(target, output, from\_logits=False, axis=-1):

"""Categorical crossentropy with integer targets.

# Arguments

target: An integer tensor.

output: A tensor resulting from a softmax

(unless `from\_logits` is True, in which

case `output` is expected to be the logits).

from\_logits: Boolean, whether `output` is the

result of a softmax, or is a tensor of logits.

axis: Int specifying the channels axis. `axis=-1`

corresponds to data format `channels\_last`,

and `axis=1` corresponds to data format

`channels\_first`.

# Returns

Output tensor.

# Raises

ValueError: if `axis` is neither -1 nor one of

the axes of `output`.

"""

return tf\_keras\_backend.sparse\_categorical\_crossentropy(

target, output, from\_logits=from\_logits, axis=axis)

def binary\_crossentropy(target, output, from\_logits=False):

"""Binary crossentropy between an output tensor and a target tensor.

# Arguments

target: A tensor with the same shape as `output`.

output: A tensor.

from\_logits: Whether `output` is expected to be a logits tensor.

By default, we consider that `output`

encodes a probability distribution.

# Returns

A tensor.

"""

return tf\_keras\_backend.binary\_crossentropy(

target, output, from\_logits=from\_logits)

def sigmoid(x):

"""Element-wise sigmoid.

# Arguments

x: A tensor or variable.

# Returns

A tensor.

{{np\_implementation}}

"""

return tf.nn.sigmoid(x)

def hard\_sigmoid(x):

"""Segment-wise linear approximation of sigmoid.

Faster than sigmoid.

Returns `0.` if `x < -2.5`, `1.` if `x > 2.5`.

In `-2.5 <= x <= 2.5`, returns `0.2 \* x + 0.5`.

# Arguments

x: A tensor or variable.

# Returns

A tensor.

{{np\_implementation}}

"""

return tf\_keras\_backend.hard\_sigmoid(x)

def tanh(x):

"""Element-wise tanh.

# Arguments

x: A tensor or variable.

# Returns

A tensor.

{{np\_implementation}}

"""

return tf.nn.tanh(x)

def dropout(x, level, noise\_shape=None, seed=None):

"""Sets entries in `x` to zero at random, while scaling the entire tensor.

# Arguments

x: tensor

level: fraction of the entries in the tensor

that will be set to 0.

noise\_shape: shape for randomly generated keep/drop flags,

must be broadcastable to the shape of `x`

seed: random seed to ensure determinism.

# Returns

A tensor.

{{np\_implementation}}

"""

if seed is None:

seed = np.random.randint(10e6)

return tf.nn.dropout(x, rate=level, noise\_shape=noise\_shape, seed=seed)

def l2\_normalize(x, axis=None):

"""Normalizes a tensor wrt the L2 norm alongside the specified axis.

# Arguments

x: Tensor or variable.

axis: axis along which to perform normalization.

# Returns

A tensor.

{{np\_implementation}}

"""

return tf.nn.l2\_normalize(x, axis=axis)

def in\_top\_k(predictions, targets, k):

"""Returns whether the `targets` are in the top `k` `predictions`.

# Arguments

predictions: A tensor of shape `(batch\_size, classes)` and type `float32`.

targets: A 1D tensor of length `batch\_size` and type `int32` or `int64`.

k: An `int`, number of top elements to consider.

# Returns

A 1D tensor of length `batch\_size` and type `bool`.

`output[i]` is `True` if `predictions[i, targets[i]]` is within top-`k`

values of `predictions[i]`.

"""

# Note that the order of the 2 first positional arguments

# has been inverted in TF 2.

return tf.nn.in\_top\_k(predictions=predictions,

targets=targets,

k=k)

# CONVOLUTIONS

def \_preprocess\_conv1d\_input(x, data\_format):

"""Transpose and cast the input before the conv1d.

# Arguments

x: input tensor.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

# Returns

A tensor.

"""

# tensorflow doesn't support float64 for conv layer before 1.8.0

if (dtype(x) == 'float64' and

StrictVersion(tf.\_\_version\_\_.split('-')[0]) < StrictVersion('1.8.0')):

x = tf.cast(x, 'float32')

tf\_data\_format = 'NWC' # to pass TF Conv2dNative operations

if data\_format == 'channels\_first':

if not \_has\_nchw\_support():

x = tf.transpose(x, (0, 2, 1)) # NCW -> NWC

else:

tf\_data\_format = 'NCW'

return x, tf\_data\_format

def \_preprocess\_conv2d\_input(x, data\_format, force\_transpose=False):

"""Transpose and cast the input before the conv2d.

# Arguments

x: input tensor.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

force\_transpose: boolean, whether force to transpose input from NCHW to NHWC

if the `data\_format` is `"channels\_first"`.

# Returns

A tensor.

"""

# tensorflow doesn't support float64 for conv layer before 1.8.0

if (dtype(x) == 'float64' and

StrictVersion(tf.\_\_version\_\_.split('-')[0]) < StrictVersion('1.8.0')):

x = tf.cast(x, 'float32')

tf\_data\_format = 'NHWC'

if data\_format == 'channels\_first':

if not \_has\_nchw\_support() or force\_transpose:

x = tf.transpose(x, (0, 2, 3, 1)) # NCHW -> NHWC

else:

tf\_data\_format = 'NCHW'

return x, tf\_data\_format

def \_preprocess\_conv3d\_input(x, data\_format):

"""Transpose and cast the input before the conv3d.

# Arguments

x: input tensor.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

# Returns

A tensor.

"""

# tensorflow doesn't support float64 for conv layer before 1.8.0

if (dtype(x) == 'float64' and

StrictVersion(tf.\_\_version\_\_.split('-')[0]) < StrictVersion('1.8.0')):

x = tf.cast(x, 'float32')

tf\_data\_format = 'NDHWC'

if data\_format == 'channels\_first':

if not \_has\_nchw\_support():

x = tf.transpose(x, (0, 2, 3, 4, 1))

else:

tf\_data\_format = 'NCDHW'

return x, tf\_data\_format

def \_preprocess\_padding(padding):

"""Convert keras' padding to tensorflow's padding.

# Arguments

padding: string, `"same"` or `"valid"`.

# Returns

a string, `"SAME"` or `"VALID"`.

# Raises

ValueError: if `padding` is invalid.

"""

if padding == 'same':

padding = 'SAME'

elif padding == 'valid':

padding = 'VALID'

else:

raise ValueError('Invalid padding: ' + str(padding))

return padding

def conv1d(x, kernel, strides=1, padding='valid',

data\_format=None, dilation\_rate=1):

"""1D convolution.

# Arguments

x: Tensor or variable.

kernel: kernel tensor.

strides: stride integer.

padding: string, `"same"`, `"causal"` or `"valid"`.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

dilation\_rate: integer dilate rate.

# Returns

A tensor, result of 1D convolution.

# Raises

ValueError: If `data\_format` is neither

`"channels\_last"` nor `"channels\_first"`.

"""

data\_format = normalize\_data\_format(data\_format)

kernel\_shape = kernel.shape.as\_list()

if padding == 'causal':

if data\_format != 'channels\_last':

raise ValueError('When using causal padding in `conv1d`, '

'`data\_format` must be "channels\_last" '

'(temporal data).')

# causal (dilated) convolution:

left\_pad = dilation\_rate \* (kernel\_shape[0] - 1)

x = temporal\_padding(x, (left\_pad, 0))

padding = 'valid'

padding = \_preprocess\_padding(padding)

x, tf\_data\_format = \_preprocess\_conv1d\_input(x, data\_format)

# TF 2 arg conversion

kwargs = {}

if \_is\_tf\_1():

kwargs['dilation\_rate'] = (dilation\_rate,)

else:

kwargs['dilations'] = (dilation\_rate,)

x = tf.nn.convolution(

x, kernel,

strides=(strides,),

padding=padding,

data\_format=tf\_data\_format,

\*\*kwargs)

if data\_format == 'channels\_first' and tf\_data\_format == 'NWC':

x = tf.transpose(x, (0, 2, 1)) # NWC -> NCW

return x

def conv2d(x, kernel, strides=(1, 1), padding='valid',

data\_format=None, dilation\_rate=(1, 1)):

"""2D convolution.

# Arguments

x: Tensor or variable.

kernel: kernel tensor.

strides: strides tuple.

padding: string, `"same"` or `"valid"`.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

Whether to use Theano or TensorFlow/CNTK data format

for inputs/kernels/outputs.

dilation\_rate: tuple of 2 integers.

# Returns

A tensor, result of 2D convolution.

# Raises

ValueError: If `data\_format` is neither

`"channels\_last"` nor `"channels\_first"`.

"""

data\_format = normalize\_data\_format(data\_format)

x, tf\_data\_format = \_preprocess\_conv2d\_input(x, data\_format)

padding = \_preprocess\_padding(padding)

# TF 2 arg conversion

kwargs = {}

if \_is\_tf\_1():

kwargs['dilation\_rate'] = dilation\_rate

else:

kwargs['dilations'] = dilation\_rate

x = tf.nn.convolution(

x, kernel,

strides=strides,

padding=padding,

data\_format=tf\_data\_format,

\*\*kwargs)

if data\_format == 'channels\_first' and tf\_data\_format == 'NHWC':

x = tf.transpose(x, (0, 3, 1, 2)) # NHWC -> NCHW

return x

def conv2d\_transpose(x, kernel, output\_shape, strides=(1, 1),

padding='valid', data\_format=None, dilation\_rate=(1, 1)):

"""2D deconvolution (i.e. transposed convolution).

# Arguments

x: Tensor or variable.

kernel: kernel tensor.

output\_shape: 1D int tensor for the output shape.

strides: strides tuple.

padding: string, `"same"` or `"valid"`.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

Whether to use Theano or TensorFlow/CNTK data format

for inputs/kernels/outputs.

dilation\_rate: tuple of 2 integers.

# Returns

A tensor, result of transposed 2D convolution.

# Raises

ValueError: If `data\_format` is neither

`"channels\_last"` nor `"channels\_first"`.

"""

data\_format = normalize\_data\_format(data\_format)

# tf.nn.atrous\_conv2d\_transpose input only supports NHWC format

if data\_format == 'channels\_first' and dilation\_rate != (1, 1):

force\_transpose = True

else:

force\_transpose = False

x, tf\_data\_format = \_preprocess\_conv2d\_input(x, data\_format, force\_transpose)

if data\_format == 'channels\_first' and tf\_data\_format == 'NHWC':

output\_shape = (output\_shape[0],

output\_shape[2],

output\_shape[3],

output\_shape[1])

if output\_shape[0] is None:

output\_shape = (shape(x)[0],) + tuple(output\_shape[1:])

output\_shape = tf.stack(list(output\_shape))

padding = \_preprocess\_padding(padding)

if tf\_data\_format == 'NHWC':

strides = (1,) + strides + (1,)

else:

strides = (1, 1) + strides

if dilation\_rate == (1, 1):

x = tf.nn.conv2d\_transpose(x, kernel, output\_shape, strides,

padding=padding,

data\_format=tf\_data\_format)

else:

assert dilation\_rate[0] == dilation\_rate[1]

x = tf.nn.atrous\_conv2d\_transpose(

x, kernel, output\_shape, dilation\_rate[0], padding)

if data\_format == 'channels\_first' and tf\_data\_format == 'NHWC':

x = tf.transpose(x, (0, 3, 1, 2)) # NHWC -> NCHW

return x

def separable\_conv1d(x, depthwise\_kernel, pointwise\_kernel, strides=1,

padding='valid', data\_format=None, dilation\_rate=1):

"""1D convolution with separable filters.

# Arguments

x: input tensor

depthwise\_kernel: convolution kernel for the depthwise convolution.

pointwise\_kernel: kernel for the 1x1 convolution.

strides: stride integer.

padding: string, `"same"` or `"valid"`.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

dilation\_rate: integer dilation rate.

# Returns

Output tensor.

# Raises

ValueError: If `data\_format` is neither

`"channels\_last"` nor `"channels\_first"`.

"""

data\_format = normalize\_data\_format(data\_format)

if isinstance(strides, int):

strides = (strides,)

if isinstance(dilation\_rate, int):

dilation\_rate = (dilation\_rate,)

x, tf\_data\_format = \_preprocess\_conv1d\_input(x, data\_format)

if tf\_data\_format == 'NWC':

tf\_data\_format = 'NHWC'

else:

tf\_data\_format = 'NCHW'

padding = \_preprocess\_padding(padding)

if tf\_data\_format == 'NHWC':

spatial\_start\_dim = 1

strides = (1,) + strides \* 2 + (1,)

else:

spatial\_start\_dim = 2

strides = (1, 1) + strides \* 2

x = tf.expand\_dims(x, spatial\_start\_dim)

depthwise\_kernel = tf.expand\_dims(depthwise\_kernel, 0)

pointwise\_kernel = tf.expand\_dims(pointwise\_kernel, 0)

dilation\_rate = (1,) + dilation\_rate

# TF 2 arg conversion

kwargs = {}

if \_is\_tf\_1():

kwargs['rate'] = dilation\_rate

else:

kwargs['dilations'] = dilation\_rate

x = tf.nn.separable\_conv2d(x, depthwise\_kernel, pointwise\_kernel,

strides=strides,

padding=padding,

data\_format=tf\_data\_format,

\*\*kwargs)

x = tf.squeeze(x, [spatial\_start\_dim])

if data\_format == 'channels\_first' and tf\_data\_format == 'NHWC':

x = tf.transpose(x, (0, 2, 1)) # NWC -> NCW

return x

def separable\_conv2d(x, depthwise\_kernel, pointwise\_kernel, strides=(1, 1),

padding='valid', data\_format=None, dilation\_rate=(1, 1)):

"""2D convolution with separable filters.

# Arguments

x: input tensor

depthwise\_kernel: convolution kernel for the depthwise convolution.

pointwise\_kernel: kernel for the 1x1 convolution.

strides: strides tuple (length 2).

padding: string, `"same"` or `"valid"`.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

dilation\_rate: tuple of integers,

dilation rates for the separable convolution.

# Returns

Output tensor.

# Raises

ValueError: If `data\_format` is neither

`"channels\_last"` nor `"channels\_first"`.

"""

data\_format = normalize\_data\_format(data\_format)

x, tf\_data\_format = \_preprocess\_conv2d\_input(x, data\_format)

padding = \_preprocess\_padding(padding)

if tf\_data\_format == 'NHWC':

strides = (1,) + strides + (1,)

else:

strides = (1, 1) + strides

# TF 2 arg conversion

kwargs = {}

if \_is\_tf\_1():

kwargs['rate'] = dilation\_rate

else:

kwargs['dilations'] = dilation\_rate

x = tf.nn.separable\_conv2d(x, depthwise\_kernel, pointwise\_kernel,

strides=strides,

padding=padding,

data\_format=tf\_data\_format,

\*\*kwargs)

if data\_format == 'channels\_first' and tf\_data\_format == 'NHWC':

x = tf.transpose(x, (0, 3, 1, 2)) # NHWC -> NCHW

return x

def depthwise\_conv2d(x, depthwise\_kernel, strides=(1, 1), padding='valid',

data\_format=None, dilation\_rate=(1, 1)):

"""2D convolution with separable filters.

# Arguments

x: input tensor

depthwise\_kernel: convolution kernel for the depthwise convolution.

strides: strides tuple (length 2).

padding: string, `"same"` or `"valid"`.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

dilation\_rate: tuple of integers,

dilation rates for the separable convolution.

# Returns

Output tensor.

# Raises

ValueError: If `data\_format` is neither

`"channels\_last"` nor `"channels\_first"`.

"""

data\_format = normalize\_data\_format(data\_format)

x, tf\_data\_format = \_preprocess\_conv2d\_input(x, data\_format)

padding = \_preprocess\_padding(padding)

if tf\_data\_format == 'NHWC':

strides = (1,) + strides + (1,)

else:

strides = (1, 1) + strides

# TF 2 arg conversion

kwargs = {}

if \_is\_tf\_1():

kwargs['rate'] = dilation\_rate

else:

kwargs['dilations'] = dilation\_rate

x = tf.nn.depthwise\_conv2d(x, depthwise\_kernel,

strides=strides,

padding=padding,

data\_format=tf\_data\_format,

\*\*kwargs)

if data\_format == 'channels\_first' and tf\_data\_format == 'NHWC':

x = tf.transpose(x, (0, 3, 1, 2)) # NHWC -> NCHW

return x

def conv3d(x, kernel, strides=(1, 1, 1), padding='valid',

data\_format=None, dilation\_rate=(1, 1, 1)):

"""3D convolution.

# Arguments

x: Tensor or variable.

kernel: kernel tensor.

strides: strides tuple.

padding: string, `"same"` or `"valid"`.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

Whether to use Theano or TensorFlow/CNTK data format

for inputs/kernels/outputs.

dilation\_rate: tuple of 3 integers.

# Returns

A tensor, result of 3D convolution.

# Raises

ValueError: If `data\_format` is neither

`"channels\_last"` nor `"channels\_first"`.

"""

data\_format = normalize\_data\_format(data\_format)

x, tf\_data\_format = \_preprocess\_conv3d\_input(x, data\_format)

padding = \_preprocess\_padding(padding)

# TF 2 arg conversion

kwargs = {}

if \_is\_tf\_1():

kwargs['dilation\_rate'] = dilation\_rate

else:

kwargs['dilations'] = dilation\_rate

x = tf.nn.convolution(

x, kernel,

strides=strides,

padding=padding,

data\_format=tf\_data\_format,

\*\*kwargs)

if data\_format == 'channels\_first' and tf\_data\_format == 'NDHWC':

x = tf.transpose(x, (0, 4, 1, 2, 3))

return x

def conv3d\_transpose(x, kernel, output\_shape, strides=(1, 1, 1),

padding='valid', data\_format=None):

"""3D deconvolution (i.e. transposed convolution).

# Arguments

x: input tensor.

kernel: kernel tensor.

output\_shape: 1D int tensor for the output shape.

strides: strides tuple.

padding: string, "same" or "valid".

data\_format: string, `"channels\_last"` or `"channels\_first"`.

Whether to use Theano or TensorFlow/CNTK data format

for inputs/kernels/outputs.

# Returns

A tensor, result of transposed 3D convolution.

# Raises

ValueError: If `data\_format` is neither

`"channels\_last"` nor `"channels\_first"`.

"""

data\_format = normalize\_data\_format(data\_format)

if isinstance(output\_shape, (tuple, list)):

output\_shape = tf.stack(output\_shape)

x, tf\_data\_format = \_preprocess\_conv3d\_input(x, data\_format)

if data\_format == 'channels\_first' and tf\_data\_format == 'NDHWC':

output\_shape = (output\_shape[0],

output\_shape[2],

output\_shape[3],

output\_shape[4],

output\_shape[1])

if output\_shape[0] is None:

output\_shape = (tf.shape(x)[0],) + tuple(output\_shape[1:])

output\_shape = tf.stack(list(output\_shape))

padding = \_preprocess\_padding(padding)

if tf\_data\_format == 'NDHWC':

strides = (1,) + strides + (1,)

else:

strides = (1, 1) + strides

x = tf.nn.conv3d\_transpose(x, kernel, output\_shape, strides,

padding=padding,

data\_format=tf\_data\_format)

if data\_format == 'channels\_first' and tf\_data\_format == 'NDHWC':

x = tf.transpose(x, (0, 4, 1, 2, 3))

return x

def pool2d(x, pool\_size, strides=(1, 1),

padding='valid', data\_format=None,

pool\_mode='max'):

"""2D Pooling.

# Arguments

x: Tensor or variable.

pool\_size: tuple of 2 integers.

strides: tuple of 2 integers.

padding: string, `"same"` or `"valid"`.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

pool\_mode: string, `"max"` or `"avg"`.

# Returns

A tensor, result of 2D pooling.

# Raises

ValueError: if `data\_format` is

neither `"channels\_last"` or `"channels\_first"`.

ValueError: if `pool\_mode` is neither `"max"` or `"avg"`.

"""

data\_format = normalize\_data\_format(data\_format)

x, tf\_data\_format = \_preprocess\_conv2d\_input(x, data\_format)

padding = \_preprocess\_padding(padding)

if tf\_data\_format == 'NHWC':

strides = (1,) + strides + (1,)

pool\_size = (1,) + pool\_size + (1,)

else:

strides = (1, 1) + strides

pool\_size = (1, 1) + pool\_size

if pool\_mode == 'max':

x = tf.nn.max\_pool(x, pool\_size, strides,

padding=padding,

data\_format=tf\_data\_format)

elif pool\_mode == 'avg':

x = tf.nn.avg\_pool(x, pool\_size, strides,

padding=padding,

data\_format=tf\_data\_format)

else:

raise ValueError('Invalid pool\_mode: ' + str(pool\_mode))

if data\_format == 'channels\_first' and tf\_data\_format == 'NHWC':

x = tf.transpose(x, (0, 3, 1, 2)) # NHWC -> NCHW

return x

def pool3d(x, pool\_size, strides=(1, 1, 1), padding='valid',

data\_format=None, pool\_mode='max'):

"""3D Pooling.

# Arguments

x: Tensor or variable.

pool\_size: tuple of 3 integers.

strides: tuple of 3 integers.

padding: string, `"same"` or `"valid"`.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

pool\_mode: string, `"max"` or `"avg"`.

# Returns

A tensor, result of 3D pooling.

# Raises

ValueError: if `data\_format` is

neither `"channels\_last"` or `"channels\_first"`.

ValueError: if `pool\_mode` is neither `"max"` or `"avg"`.

"""

data\_format = normalize\_data\_format(data\_format)

x, tf\_data\_format = \_preprocess\_conv3d\_input(x, data\_format)

padding = \_preprocess\_padding(padding)

if tf\_data\_format == 'NDHWC':

strides = (1,) + strides + (1,)

pool\_size = (1,) + pool\_size + (1,)

else:

strides = (1, 1) + strides

pool\_size = (1, 1) + pool\_size

if pool\_mode == 'max':

x = tf.nn.max\_pool3d(x, pool\_size, strides,

padding=padding,

data\_format=tf\_data\_format)

elif pool\_mode == 'avg':

x = tf.nn.avg\_pool3d(x, pool\_size, strides,

padding=padding,

data\_format=tf\_data\_format)

else:

raise ValueError('Invalid pool\_mode: ' + str(pool\_mode))

if data\_format == 'channels\_first' and tf\_data\_format == 'NDHWC':

x = tf.transpose(x, (0, 4, 1, 2, 3))

return x

def local\_conv1d(inputs, kernel, kernel\_size, strides, data\_format=None):

"""Apply 1D conv with un-shared weights.

# Arguments

inputs: 3D tensor with shape: (batch\_size, steps, input\_dim)

kernel: the unshared weight for convolution,

with shape (output\_length, feature\_dim, filters)

kernel\_size: a tuple of a single integer,

specifying the length of the 1D convolution window

strides: a tuple of a single integer,

specifying the stride length of the convolution

data\_format: the data format, channels\_first or channels\_last

# Returns

the tensor after 1d conv with un-shared weights,

with shape (batch\_size, output\_length, filters)

# Raises

ValueError: If `data\_format` is neither

`"channels\_last"` nor `"channels\_first"`.

"""

data\_format = normalize\_data\_format(data\_format)

stride = strides[0]

kernel\_shape = int\_shape(kernel)

output\_length, feature\_dim, filters = kernel\_shape

xs = []

for i in range(output\_length):

slice\_length = py\_slice(i \* stride,

i \* stride + kernel\_size[0])

xs.append(reshape(inputs[:, slice\_length, :],

(1, -1, feature\_dim)))

x\_aggregate = concatenate(xs, axis=0)

# Shape: `(output\_length, batch\_size, filters)`.

output = batch\_dot(x\_aggregate, kernel)

return permute\_dimensions(output, (1, 0, 2))

def local\_conv2d(inputs,

kernel,

kernel\_size,

strides,

output\_shape,

data\_format=None):

"""Apply 2D conv with un-shared weights.

# Arguments

inputs: 4D tensor with shape:

(batch\_size, filters, new\_rows, new\_cols)

if data\_format='channels\_first'

or 4D tensor with shape:

(batch\_size, new\_rows, new\_cols, filters)

if data\_format='channels\_last'.

kernel: the unshared weight for convolution,

with shape (output\_items, feature\_dim, filters)

kernel\_size: a tuple of 2 integers, specifying the

width and height of the 2D convolution window.

strides: a tuple of 2 integers, specifying the strides

of the convolution along the width and height.

output\_shape: a tuple with (output\_row, output\_col)

data\_format: the data format, channels\_first or channels\_last

# Returns

A 4d tensor with shape:

(batch\_size, filters, new\_rows, new\_cols)

if data\_format='channels\_first'

or 4D tensor with shape:

(batch\_size, new\_rows, new\_cols, filters)

if data\_format='channels\_last'.

# Raises

ValueError: if `data\_format` is neither

`channels\_last` or `channels\_first`.

"""

data\_format = normalize\_data\_format(data\_format)

stride\_row, stride\_col = strides

output\_row, output\_col = output\_shape

kernel\_shape = int\_shape(kernel)

\_, feature\_dim, filters = kernel\_shape

xs = []

for i in range(output\_row):

for j in range(output\_col):

slice\_row = py\_slice(i \* stride\_row,

i \* stride\_row + kernel\_size[0])

slice\_col = py\_slice(j \* stride\_col,

j \* stride\_col + kernel\_size[1])

if data\_format == 'channels\_first':

xs.append(reshape(inputs[:, :, slice\_row, slice\_col],

(1, -1, feature\_dim)))

else:

xs.append(reshape(inputs[:, slice\_row, slice\_col, :],

(1, -1, feature\_dim)))

x\_aggregate = concatenate(xs, axis=0)

output = batch\_dot(x\_aggregate, kernel)

output = reshape(output,

(output\_row, output\_col, -1, filters))

if data\_format == 'channels\_first':

output = permute\_dimensions(output, (2, 3, 0, 1))

else:

output = permute\_dimensions(output, (2, 0, 1, 3))

return output

def bias\_add(x, bias, data\_format=None):

"""Adds a bias vector to a tensor.

# Arguments

x: Tensor or variable.

bias: Bias tensor to add.

data\_format: string, `"channels\_last"` or `"channels\_first"`.

# Returns

Output tensor.

# Raises

ValueError: In one of the two cases below:

1. invalid `data\_format` argument.

2. invalid bias shape.

the bias should be either a vector or

a tensor with ndim(x) - 1 dimension

{{np\_implementation}}

"""

data\_format = normalize\_data\_format(data\_format)

bias\_shape = int\_shape(bias)

if len(bias\_shape) != 1 and len(bias\_shape) != ndim(x) - 1:

raise ValueError('Unexpected bias dimensions %d, '

'expect to be 1 or %d dimensions'

% (len(bias\_shape), ndim(x)))

if ndim(x) == 5:

if len(bias\_shape) == 1:

new\_shape = (1, 1, 1, 1, bias\_shape[0])

else:

new\_shape = (1,) + bias\_shape

new\_shape = transpose\_shape(new\_shape, data\_format,

spatial\_axes=(1, 2, 3))

x = x + reshape(bias, new\_shape)

elif ndim(x) == 4:

if data\_format == 'channels\_first':

if len(bias\_shape) == 1:

if \_has\_nchw\_support():

x = tf.nn.bias\_add(x, bias,

data\_format='NCHW')

else:

x = x + reshape(bias, (1, bias\_shape[0], 1, 1))

else:

x = x + reshape(bias, (1, bias\_shape[2]) + bias\_shape[:2])

elif data\_format == 'channels\_last':

if len(bias\_shape) == 1:

x = tf.nn.bias\_add(x, bias,

data\_format='NHWC')

else:

x = x + reshape(bias, (1,) + bias\_shape)

elif ndim(x) == 3:

if len(bias\_shape) == 1:

new\_shape = (1, 1, bias\_shape[0])

else:

new\_shape = (1,) + bias\_shape

new\_shape = transpose\_shape(new\_shape, data\_format,

spatial\_axes=(1,))

x = x + reshape(bias, new\_shape)

else:

x = tf.nn.bias\_add(x, bias)

return x

# RANDOMNESS

def random\_normal(shape, mean=0.0, stddev=1.0, dtype=None, seed=None):

"""Returns a tensor with normal distribution of values.

# Arguments

shape: A tuple of integers, the shape of tensor to create.

mean: A float, mean of the normal distribution to draw samples.

stddev: A float, standard deviation of the normal distribution

to draw samples.

dtype: String, dtype of returned tensor.

seed: Integer, random seed.

# Returns

A tensor.

"""

if dtype is None:

dtype = floatx()

if seed is None:

seed = np.random.randint(10e6)

if py\_any(list(is\_symbolic(x) for x in (shape, mean, stddev))):

with get\_graph().as\_default():

return tf\_keras\_backend.random\_normal(

shape, mean=mean, stddev=stddev, dtype=dtype, seed=seed)

with tf\_ops.init\_scope():

return tf\_keras\_backend.random\_normal(

shape, mean=mean, stddev=stddev, dtype=dtype, seed=seed)

def random\_uniform(shape, minval=0.0, maxval=1.0, dtype=None, seed=None):

"""Returns a tensor with uniform distribution of values.

# Arguments

shape: A tuple of integers, the shape of tensor to create.

minval: A float, lower boundary of the uniform distribution

to draw samples.

maxval: A float, upper boundary of the uniform distribution

to draw samples.

dtype: String, dtype of returned tensor.

seed: Integer, random seed.

# Returns

A tensor.

"""

if dtype is None:

dtype = floatx()

if seed is None:

seed = np.random.randint(10e6)

if py\_any(list(is\_symbolic(x) for x in (shape, minval, maxval))):

with get\_graph().as\_default():

return tf\_keras\_backend.random\_uniform(

shape, minval=minval, maxval=maxval, dtype=dtype, seed=seed)

with tf\_ops.init\_scope():

return tf\_keras\_backend.random\_uniform(

shape, minval=minval, maxval=maxval, dtype=dtype, seed=seed)

def random\_binomial(shape, p=0.0, dtype=None, seed=None):

"""Returns a tensor with random binomial distribution of values.

# Arguments

shape: A tuple of integers, the shape of tensor to create.

p: A float, `0. <= p <= 1`, probability of binomial distribution.

dtype: String, dtype of returned tensor.

seed: Integer, random seed.

# Returns

A tensor.

"""

if dtype is None:

dtype = floatx()

if seed is None:

seed = np.random.randint(10e6)

if py\_any(list(is\_symbolic(x) for x in (shape, p))):

with get\_graph().as\_default():

return tf\_keras\_backend.random\_binomial(

shape, p=p, dtype=dtype, seed=seed)

with tf\_ops.init\_scope():

return tf\_keras\_backend.random\_binomial(

shape, p=p, dtype=dtype, seed=seed)

def truncated\_normal(shape, mean=0.0, stddev=1.0, dtype=None, seed=None):

"""Returns a tensor with truncated random normal distribution of values.

The generated values follow a normal distribution

with specified mean and standard deviation,

except that values whose magnitude is more than

two standard deviations from the mean are dropped and re-picked.

# Arguments

shape: A tuple of integers, the shape of tensor to create.

mean: Mean of the values.

stddev: Standard deviation of the values.

dtype: String, dtype of returned tensor.

seed: Integer, random seed.

# Returns

A tensor.

"""

if dtype is None:

dtype = floatx()

if seed is None:

seed = np.random.randint(10e6)

if py\_any(list(is\_symbolic(x) for x in (shape, mean, stddev))):

with get\_graph().as\_default():

return tf\_keras\_backend.truncated\_normal(

shape, mean=mean, stddev=stddev, dtype=dtype, seed=seed)

with tf\_ops.init\_scope():

return tf\_keras\_backend.truncated\_normal(

shape, mean=mean, stddev=stddev, dtype=dtype, seed=seed)

# CTC

# TensorFlow has a native implementation, but it uses sparse tensors

# and therefore requires a wrapper for Keras. The functions below convert

# dense to sparse tensors and also wraps up the beam search code that is

# in TensorFlow's CTC implementation

def ctc\_label\_dense\_to\_sparse(labels, label\_lengths):

"""Converts CTC labels from dense to sparse.

# Arguments

labels: dense CTC labels.

label\_lengths: length of the labels.

# Returns

A sparse tensor representation of the labels.

"""

label\_shape = tf.shape(labels)

num\_batches\_tns = tf.stack([label\_shape[0]])

max\_num\_labels\_tns = tf.stack([label\_shape[1]])

def range\_less\_than(\_, current\_input):

return tf.expand\_dims(tf.range(label\_shape[1]), 0) < tf.fill(

max\_num\_labels\_tns, current\_input)

init = tf.cast(tf.fill([1, label\_shape[1]], 0), tf.bool)

dense\_mask = functional\_ops.scan(range\_less\_than, label\_lengths,

initializer=init, parallel\_iterations=1)

dense\_mask = dense\_mask[:, 0, :]

label\_array = tf.reshape(tf.tile(tf.range(label\_shape[1]), num\_batches\_tns),

label\_shape)

label\_ind = tf.boolean\_mask(label\_array, dense\_mask)

tmp = tf.tile(tf.range(label\_shape[0]), max\_num\_labels\_tns)

batch\_array = tf.transpose(tf.reshape(tmp, reverse(label\_shape, 0)))

batch\_ind = tf.boolean\_mask(batch\_array, dense\_mask)

indices = concatenate([batch\_ind, label\_ind], axis=0)

indices = tf.transpose(tf.reshape(indices, [2, -1]))

vals\_sparse = tf.gather\_nd(labels, indices)

indices = tf.cast(indices, tf.int64)

label\_shape = tf.cast(label\_shape, tf.int64)

return tf.SparseTensor(indices, vals\_sparse, label\_shape)

def ctc\_batch\_cost(y\_true, y\_pred, input\_length, label\_length):

"""Runs CTC loss algorithm on each batch element.

# Arguments

y\_true: tensor `(samples, max\_string\_length)`

containing the truth labels.

y\_pred: tensor `(samples, time\_steps, num\_categories)`

containing the prediction, or output of the softmax.

input\_length: tensor `(samples, 1)` containing the sequence length for

each batch item in `y\_pred`.

label\_length: tensor `(samples, 1)` containing the sequence length for

each batch item in `y\_true`.

# Returns

Tensor with shape (samples,1) containing the

CTC loss of each element.

"""

label\_length = tf.cast(tf.squeeze(label\_length, axis=-1), tf.int32)

input\_length = tf.cast(tf.squeeze(input\_length, axis=-1), tf.int32)

sparse\_labels = tf.cast(

ctc\_label\_dense\_to\_sparse(y\_true, label\_length), tf.int32)

y\_pred = tf\_math\_ops.log(tf.transpose(y\_pred, perm=[1, 0, 2]) + epsilon())

return tf.expand\_dims(ctc.ctc\_loss(inputs=y\_pred,

labels=sparse\_labels,

sequence\_length=input\_length), 1)

def ctc\_decode(y\_pred, input\_length, greedy=True, beam\_width=100,

top\_paths=1, merge\_repeated=False):

"""Decodes the output of a softmax.

Can use either greedy search (also known as best path)

or a constrained dictionary search.

# Arguments

y\_pred: tensor `(samples, time\_steps, num\_categories)`

containing the prediction, or output of the softmax.

input\_length: tensor `(samples, )` containing the sequence length for

each batch item in `y\_pred`.

greedy: perform much faster best-path search if `True`.

This does not use a dictionary.

beam\_width: if `greedy` is `False`: a beam search decoder will be used

with a beam of this width.

top\_paths: if `greedy` is `False`,

how many of the most probable paths will be returned.

merge\_repeated: if `greedy` is `False`,

merge repeated classes in the output beams.

# Returns

Tuple:

List: if `greedy` is `True`, returns a list of one element that

contains the decoded sequence.

If `False`, returns the `top\_paths` most probable

decoded sequences.

Important: blank labels are returned as `-1`.

Tensor `(top\_paths, )` that contains

the log probability of each decoded sequence.

"""

y\_pred = tf\_math\_ops.log(tf.transpose(y\_pred, perm=[1, 0, 2]) + epsilon())

input\_length = tf.cast(input\_length, tf.int32)

if greedy:

(decoded, log\_prob) = ctc.ctc\_greedy\_decoder(

inputs=y\_pred,

sequence\_length=input\_length)

else:

(decoded, log\_prob) = ctc.ctc\_beam\_search\_decoder(

inputs=y\_pred,

sequence\_length=input\_length, beam\_width=beam\_width,

top\_paths=top\_paths, merge\_repeated=merge\_repeated)

decoded\_dense = []

for st in decoded:

dense\_tensor = tf.sparse.to\_dense(st, default\_value=-1)

decoded\_dense.append(dense\_tensor)

return decoded\_dense, log\_prob

def control\_dependencies(control\_inputs):

"""A context manager that specifies control dependencies.

# Arguments

control\_inputs: A list of Operation or Tensor objects

which must be executed

or computed before running the operations defined in the context.

Can also be None to clear the control dependencies.

# Returns

A context manager.

"""

return tf.control\_dependencies(control\_inputs)

# HIGH ORDER FUNCTIONS

def map\_fn(fn, elems, name=None, dtype=None):

"""Map the function fn over the elements elems and return the outputs.

# Arguments

fn: Callable that will be called upon each element in elems

elems: tensor

name: A string name for the map node in the graph

dtype: Output data type.

# Returns

Tensor with dtype `dtype`.

"""

return tf.map\_fn(fn, elems, name=name, dtype=dtype)

def foldl(fn, elems, initializer=None, name=None):

"""Reduce elems using fn to combine them from left to right.

# Arguments

fn: Callable that will be called upon each element in elems and an

accumulator, for instance `lambda acc, x: acc + x`

elems: tensor

initializer: The first value used (`elems[0]` in case of None)

name: A string name for the foldl node in the graph

# Returns

Tensor with same type and shape as `initializer`.

"""

return tf.foldl(fn, elems, initializer=initializer, name=name)

def foldr(fn, elems, initializer=None, name=None):

"""Reduce elems using fn to combine them from right to left.

# Arguments

fn: Callable that will be called upon each element in elems and an

accumulator, for instance `lambda acc, x: acc + x`

elems: tensor

initializer: The first value used (`elems[-1]` in case of None)

name: A string name for the foldr node in the graph

# Returns

Tensor with same type and shape as `initializer`.

"""

return tf.foldr(fn, elems, initializer=initializer, name=name)