Keras backends

What is a "backend"?

Keras is a model-level library, providing high-level building blocks for developing deep learning models. It does not handle itself low-level operations such as tensor products, convolutions and so on. Instead, it relies on a specialized, well-optimized tensor manipulation library to do so, serving as the "backend engine" of Keras. Rather than picking one single tensor library and making the implementation of Keras tied to that library, Keras handles the problem in a modular way, and several different backend engines can be plugged seamlessly into Keras.

At this time, Keras has two backend implementations available: the **Theano** backend and the **TensorFlow** backend.

- Theano is an open-source symbolic tensor manipulation framework developed by LISA/MILA Lab at Université de Montréal.
- TensorFlow is an open-source symbolic tensor manipulation framework developed by Google, Inc.

Switching from one backend to another

If you have run Keras at least once, you will find the Keras configuration file at:

`~/.keras/keras.json`

If it isn't there, you can create it.

It probably looks like this:

```
{"epsilon": 1e-07, "floatx": "float32", "backend": "theano"}
```

Simply change the field `backend` to either `"theano"` or `"tensorflow"`, and Keras will use the new configuration next time you run any Keras code.

You can also define the environment variable `KERAS_BACKEND` and this will override what is defined in your config file :

```
KERAS_BACKEND=tensorflow python -c "from keras import backend; print(backend._BACKEND)"
Using TensorFlow backend.
tensorflow
```

Using the abstract Keras backend to write new code

If you want the Keras modules you write to be compatible with both Theano and TensorFlow, you have to write them via the abstract Keras backend API. Here's an intro.

You can import the backend module via:

```
from keras import backend as K
```

The code below instantiates an input placeholder. It's equivalent to `tf.placeholder()` or `T.matrix()`, `T.tensor3()`, etc.

```
input = K.placeholder(shape=(2, 4, 5))
# also works:
input = K.placeholder(shape=(None, 4, 5))
# also works:
input = K.placeholder(ndim=3)
```

The code below instantiates a shared variable. It's equivalent to `tf.variable()` or `theano.shared()`.

```
val = np.random.random((3, 4, 5))
var = K.variable(value=val)

# all-zeros variable:
var = K.zeros(shape=(3, 4, 5))
# all-ones:
var = K.ones(shape=(3, 4, 5))
```

Most tensor operations you will need can be done as you would in TensorFlow or Theano:

```
a = b + c * K.abs(d)
c = K.dot(a, K.transpose(b))
a = K.sum(b, axis=2)
a = K.softmax(b)
a = concatenate([b, c], axis=-1)
# etc...
```

Backend functions

epsilon

```
epsilon()
```

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

---

set_epsilon

```
set_epsilon(e)
```

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

---

floatx

```
floatx()
```

Returns the default float type, as a string (e.g. 'float16', 'float32', 'float64').

---

cast_to_floatx

```
cast_to_floatx(x)
```

Cast a Numpy array to floatx.

---

image_dim_ordering

```
image_dim_ordering()
```

Returns the image dimension ordering convention ('th' or 'tf').

---

set_image_dim_ordering

```
set_image_dim_ordering(dim_ordering)
```

Sets the value of the image dimension ordering convention ('th' or 'tf').

---

learning_phase

```
learning_phase()
```

Returns the learning phase flag.

The learning phase flag is an integer 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.

---

manual_variable_initialization

```
manual_variable_initialization(value)
```

Whether variables should be initialized as they are instantiated (default), or if the user should handle the initialization (e.g. via tf.initialize_all_variables()).

---

variable

```
variable(value, dtype='float32', name=None)
```

Instantiates a tensor.

Arguments

- **value**: numpy array, initial value of the tensor.
- **dtype**: tensor type.
- **name**: optional name string for the tensor.

Returns

Tensor variable instance.

---

placeholder

```
placeholder(shape=None, ndim=None, dtype='float32', name=None)
```

Instantiates a placeholder.

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.

- **name**: optional name string for the placeholder.

Returns

Placeholder tensor instance.

---

shape

```
shape(x)
```

Returns the symbolic shape of a tensor.

---

int_shape

```
int_shape(x)
```

Returns the shape of a tensor as a tuple of integers or None entries. Note that this function only works with TensorFlow.

---

ndim

```
ndim(x)
```

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

---

dtype

```
dtype(x)
```

Returns the dtype of a tensor, as a string.

---

eval

```
eval(x)
```

Evaluates the value of a tensor. Returns a Numpy array.

---

zeros

```
zeros(shape, dtype='float32', name=None)
```

Instantiates an all-zeros tensor variable.

---

ones

```
ones(shape, dtype='float32', name=None)
```

Instantiates an all-ones tensor variable.

---

eye

```
eye(size, dtype='float32', name=None)
```

Instantiate an identity matrix.

---

zeros_like

```
zeros_like(x, name=None)
```

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

---

ones_like

```
ones_like(x, name=None)
```

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

---

count_params

```
count_params(x)
```

Returns the number of scalars in a tensor.

---

cast

```
cast(x, dtype)
```

Casts a tensor to a different dtype.

---

dot

```
dot(x, y)
```

Multiplies 2 tensors. When attempting to multiply a ND tensor with a ND tensor, reproduces the Theano behavior (e.g. (2, 3).(4, 3, 5) = (2, 4, 5))

---

batch_dot

```
batch_dot(x, y, axes=None)
```

Batchwise dot product.

batch_dot results in a tensor 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, y: tensors with ndim >= 2 - **axes**: list (or single) int with target dimensions

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.

Shape inference: Let x's shape be (100, 20) and y's shape be (100, 30, 20). If dot_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)

transpose

```
transpose(x)
```

Transposes a matrix.

---

gather

```
gather(reference, indices)
```

Retrieves the vectors of indices `indices` in the 2D tensor `reference`.

Arguments

- **reference**: a 2D tensor.
- **indices**: an int tensor of indices.

Returns

A 3D tensor of same type as `reference`.

---

max

```
max(x, axis=None, keepdims=False)
```

Maximum value in a tensor.

---

min

```
min(x, axis=None, keepdims=False)
```

Minimum value in a tensor.

---

sum

```
sum(x, axis=None, keepdims=False)
```

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

---

prod

```
prod(x, axis=None, keepdims=False)
```

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

---

var

```
var(x, axis=None, keepdims=False)
```

Variance of a tensor, alongside the specified axis.

---

std

```
std(x, axis=None, keepdims=False)
```

Standard deviation of a tensor, alongside the specified axis.

---

mean

```
mean(x, axis=None, keepdims=False)
```

Mean of a tensor, alongside the specified axis.

---

any

```
any(x, axis=None, keepdims=False)
```

Bitwise reduction (logical OR).

Returns an uint8 tensor (0s and 1s).

---

all

```
all(x, axis=None, keepdims=False)
```

Bitwise reduction (logical AND).

Returns an uint8 tensor

argmax

```
argmax(x, axis=-1)
```

Returns the index of the maximum value along a tensor axis.

---

argmin

```
argmin(x, axis=-1)
```

Returns the index of the minimum value along a tensor axis.

---

square

```
square(x)
```

Element-wise square.

---

abs

```
abs(x)
```

Element-wise absolute value.

---

sqrt

```
sqrt(x)
```

Element-wise square root.

---

exp

```
exp(x)
```

Element-wise exponential.

---

log

```
log(x)
```

Element-wise log.

---

round

```
round(x)
```

Element-wise rounding to the closest integer.

---

sign

```
sign(x)
```

Element-wise sign.

---

pow

```
pow(x, a)
```

Element-wise exponentiation.

---

clip

```
clip(x, min_value, max_value)
```

Element-wise value clipping.

---

equal

```
equal(x, y)
```

Element-wise equality between two tensors. Returns a bool tensor.

---

not_equal

```
not_equal(x, y)
```

Element-wise inequality between two tensors. Returns a bool tensor.

---

greater

```
greater(x, y)
```

Element-wise truth value of (x > y). Returns a bool tensor.

---

greater_equal

```
greater_equal(x, y)
```

Element-wise truth value of (x >= y). Returns a bool tensor.

---

lesser

```
lesser(x, y)
```

Element-wise truth value of (x < y). Returns a bool tensor.

---

lesser_equal

```
lesser_equal(x, y)
```

Element-wise truth value of (x <= y). Returns a bool tensor.

---

maximum

```
maximum(x, y)
```

Element-wise maximum of two tensors.

---

minimum

```
minimum(x, y)
```

Element-wise minimum of two tensors.

---

sin

```
sin(x)
```

Computes sin of x element-wise.

---

cos

```
cos(x)
```

Computes cos of x element-wise.

---

normalize_batch_in_training

```
normalize_batch_in_training(x, gamma, beta, reduction_axes, epsilon=0.0001)
```

Compute mean and std for batch then apply batch_normalization on batch.

---

batch_normalization

```
batch_normalization(x, mean, std, beta, gamma, epsilon=0.0001)
```

Apply batch normalization on x given mean, std, beta and gamma.

---

concatenate

```
concatenate(tensors, axis=-1)
```

Concantes a list of tensors alongside the specified axis.

---

reshape

```
reshape(x, shape)
```

Reshapes a tensor to the specified shape.

---

permute_dimensions

```
permute_dimensions(x, pattern)
```

Permutes axes in a tensor.

Arguments

- **pattern**: should be a tuple of dimension indices, e.g. (0, 2, 1).

---

resize_images

```
resize_images(X, height_factor, width_factor, dim_ordering)
```

Resizes the images contained in a 4D tensor of shape - [batch, channels, height, width] (for 'th' dim_ordering) - [batch, height, width, channels] (for 'tf' dim_ordering) by a factor of (height_factor, width_factor). Both factors should be positive integers.

---

resize_volumes

```
resize_volumes(X, depth_factor, height_factor, width_factor, dim_ordering)
```

Resize the volume contained in a 5D tensor of shape - [batch, channels, depth, height, width] (for 'th' dim_ordering) - [batch, depth, height, width, channels] (for 'tf' dim_ordering) by a factor of (depth_factor, height_factor, width_factor). All three factors should be positive integers.

---

repeat_elements

```
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=1, the output will have shape (s1, s2 * rep, s3)

---

repeat

```
repeat(x, n)
```

Repeats a 2D tensor:

if x has shape (samples, dim) and n=2, the output will have shape (samples, 2, dim)

---

batch_flatten

```
batch_flatten(x)
```

Turn a n-D tensor into a 2D tensor where the first dimension is conserved.

---

expand_dims

```
expand_dims(x, dim=-1)
```

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

---

squeeze

```
squeeze(x, axis)
```

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

---

temporal_padding

```
temporal_padding(x, padding=1)
```

Pads the middle dimension of a 3D tensor with "padding" zeros left and right.

---

spatial_2d_padding

```
spatial_2d_padding(x, padding=(1, 1), dim_ordering='th')
```

Pads the 2nd and 3rd dimensions of a 4D tensor with "padding[0]" and "padding[1]" (resp.) zeros left and right.

---

spatial_3d_padding

```
spatial_3d_padding(x, padding=(1, 1, 1), dim_ordering='th')
```

Pads 5D tensor with zeros for the depth, height, width dimension with "padding[0]", "padding[1]" and "padding[2]" (resp.) zeros left and right

For 'tf' dim_ordering, the 2nd, 3rd and 4th dimension will be padded. For 'th' dim_ordering, the 3rd, 4th and 5th dimension will be padded.

---

get_value

```
get_value(x)
```

Returns the value of a tensor variable, as a Numpy array.

---

batch_get_value

```
batch_get_value(xs)
```

Returns the value of more than one tensor variable, as a list of Numpy arrays.

---

set_value

```
set_value(x, value)
```

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

---

batch_set_value

```
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.

---

print_tensor

```
print_tensor(x, message='')
```

Print the message and the tensor when evaluated and return the same tensor.

---

function

```
function(inputs, outputs, updates=[])
```

Instantiates a Keras function.

Arguments

- **inputs**: list of placeholder/variable tensors.
- **outputs**: list of output tensors.
- **updates**: list of update tuples (old_tensor, new_tensor).

---

gradients

```
gradients(loss, variables)
```

Returns the gradients of `variables` (list of tensor variables) with regard to `loss`.

---

stop_gradient

```
stop_gradient(variables)
```

Returns `variables` but with zero gradient with respect to every other variables.

---

rnn

```
rnn(step_function, inputs, initial_states, go_backwards=False, mask=None, constants=None, unrol
```

Iterates over the time dimension of a tensor.

Arguments

- **inputs**: tensor of temporal data of shape (samples, time, ...) (at least 3D).
- **step_function**:
- **Parameters**:
  - **input**: tensor with shape (samples, ...) (no time dimension), representing input for the batch of samples at a certain time step.
  - **states**: list of tensors.
- **Returns**:
  - **output**: tensor with shape (samples, ...) (no time dimension),
  - **new_states**: list of tensors, same length and shapes as 'states'.

- **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.
- **mask**: binary tensor with shape (samples, time, 1), with a zero for every element that is masked.
- **constants**: a list of constant values passed at each step.
- **unroll**: with TensorFlow the RNN is always unrolled, but with Theano you can use this boolean flag to unroll the RNN.
- **input_length**: not relevant in the TensorFlow implementation. Must be specified if using unrolling with Theano.

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, ...).

---

switch

```
switch(condition, then_expression, else_expression)
```

Switches between two operations depending on a scalar value (int or bool). Note that both `then_expression` and `else_expression` should be symbolic tensors of the *same shape*.

Arguments

- **condition**: scalar tensor.
- **then_expression**: TensorFlow operation.
- **else_expression**: TensorFlow operation.

---

in_train_phase

```
in_train_phase(x, alt)
```

Selects `x` in train phase, and `alt` otherwise. Note that `alt` should have the *same shape* as `x`.

---

in_test_phase

```
in_test_phase(x, alt)
```

Selects `x` in test phase, and `alt` otherwise. Note that `alt` should have the *same shape* as `x`.

---

relu

```
relu(x, alpha=0.0, max_value=None)
```

Rectified linear unit

Arguments

- **alpha**: slope of negative section.
- **max_value**: saturation threshold.

---

softmax

```
softmax(x)
```

Softmax of a tensor.

---

softplus

```
softplus(x)
```

Softplus of a tensor.

---

categorical_crossentropy

```
categorical_crossentropy(output, target, from_logits=False)
```

Categorical crossentropy between an output tensor and a target tensor, where the target is a tensor of the same shape as the output.

---

sparse_categorical_crossentropy

```
sparse_categorical_crossentropy(output, target, from_logits=False)
```

Categorical crossentropy between an output tensor and a target tensor, where the target is an integer tensor.

---

binary_crossentropy

```
binary_crossentropy(output, target, from_logits=False)
```

Binary crossentropy between an output tensor and a target tensor.

---

sigmoid

```
sigmoid(x)
```

Element-wise sigmoid.

---

hard_sigmoid

```
hard_sigmoid(x)
```

Segment-wise linear approximation of sigmoid. Faster than sigmoid.

---

tanh

```
tanh(x)
```

Element-wise tanh.

---

dropout

```
dropout(x, level, 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
- **seed**: random seed to ensure determinism.

---

l2_normalize

```
l2_normalize(x, axis)
```

Normalizes a tensor wrt the L2 norm alongside the specified axis.

---

conv2d

```
conv2d(x, kernel, strides=(1, 1), border_mode='valid', dim_ordering='th', image_shape=None, fil
```

2D convolution.

Arguments

- **kernel**: kernel tensor.
- **strides**: strides tuple.
- **border_mode**: string, "same" or "valid".
- **dim_ordering**: "tf" or "th". Whether to use Theano or TensorFlow dimension ordering for inputs/kernels/ouputs.

---

deconv2d

```
deconv2d(x, kernel, output_shape, strides=(1, 1), border_mode='valid', dim_ordering='th', image
```

2D deconvolution (i.e. transposed convolution).

Arguments

- **x**: input tensor.
- **kernel**: kernel tensor.
- **output_shape**: 1D int tensor for the output shape.
- **strides**: strides tuple.
- **border_mode**: string, "same" or "valid".
- **dim_ordering**: "tf" or "th". Whether to use Theano or TensorFlow dimension ordering for inputs/kernels/ouputs.

---

conv3d

```
conv3d(x, kernel, strides=(1, 1, 1), border_mode='valid', dim_ordering='th', volume_shape=None,
```

3D convolution.

Arguments

- **kernel**: kernel tensor.
- **strides**: strides tuple.
- **border_mode**: string, "same" or "valid".
- **dim_ordering**: "tf" or "th". Whether to use Theano or TensorFlow dimension ordering for inputs/kernels/ouputs.

---

pool2d

```
pool2d(x, pool_size, strides=(1, 1), border_mode='valid', dim_ordering='th', pool_mode='max')
```

2D Pooling.

Arguments

- **pool_size**: tuple of 2 integers.
- **strides**: tuple of 2 integers.
- **border_mode**: one of "valid", "same".
- **dim_ordering**: one of "th", "tf".
- **pool_mode**: one of "max", "avg".

---

pool3d

```
pool3d(x, pool_size, strides=(1, 1, 1), border_mode='valid', dim_ordering='th', pool_mode='max'
```

3D Pooling.

Arguments

- **pool_size**: tuple of 3 integers.
- **strides**: tuple of 3 integers.
- **border_mode**: one of "valid", "same".
- **dim_ordering**: one of "th", "tf".
- **pool_mode**: one of "max", "avg".