Contents

Home

Keras: The Python Deep Learning library

<img src='https://s3.amazonaws.com/keras.io/img/keras-logo-2018-large-1200.png', style='max-width: 600px;'>

You have just found Keras.

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. *Being able to go from idea to result with the least possible delay is key to doing good research.*

Use Keras if you need a deep learning library that:

- Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
- Supports both convolutional networks and recurrent networks, as well as combinations of the two.
- Runs seamlessly on CPU and GPU.

Read the documentation at Keras.io.

Keras is compatible with: **Python 2.7-3.6**.

Guiding principles

- **User friendliness.** Keras is an API designed for human beings, not machines. It puts user experience front and center. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.
- **Modularity.** A model is understood as a sequence or a graph of standalone, fully-configurable modules that can be plugged together with as little restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions, regularization schemes are all standalone modules that you can combine to create new models.
- **Easy extensibility.** New modules are simple to add (as new classes and functions), and existing modules provide ample examples. To be able to easily create new modules allows for total expressiveness, making Keras suitable for advanced research.
- **Work with Python**. No separate models configuration files in a declarative format. Models are described in Python code, which is compact, easier to debug, and allows for ease of extensibility.

Getting started: 30 seconds to Keras

The core data structure of Keras is a **model**, a way to organize layers. The simplest type of model is the `Sequential` model, a linear stack of layers. For more complex architectures, you should use the Keras functional API, which allows to build arbitrary graphs of layers.

Here is the `Sequential` model:

```
from keras.models import Sequential

model = Sequential()
```

Stacking layers is as easy as `.add()`:

```
from keras.layers import Dense

model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
```

Once your model looks good, configure its learning process with `.compile()`:

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

If you need to, you can further configure your optimizer. A core principle of Keras is to make things reasonably simple, while allowing the user to be fully in control when they need to (the ultimate control being the easy extensibility of the source code).

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(lr=0.01, momentum=0.9, nesterov=True))
```

You can now iterate on your training data in batches:

```
### x_train and y_train are Numpy arrays --just like in the Scikit-Learn API.
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Alternatively, you can feed batches to your model manually:

```
model.train_on_batch(x_batch, y_batch)
```

Evaluate your performance in one line:

```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

Or generate predictions on new data:

```
classes = model.predict(x_test, batch_size=128)
```

Building a question answering system, an image classification model, a Neural Turing Machine, or any other model is just as fast. The ideas behind deep learning are simple, so why should their implementation be painful?

For a more in-depth tutorial about Keras, you can check out:

- Getting started with the Sequential model
- Getting started with the functional API

In the examples folder of the repository, you will find more advanced models: question-answering with memory networks, text generation with stacked LSTMs, etc.

Installation

Before installing Keras, please install one of its backend engines: TensorFlow, Theano, or CNTK. We recommend the TensorFlow backend.

- TensorFlow installation instructions.
- Theano installation instructions.
- CNTK installation instructions.

You may also consider installing the following **optional dependencies**:

- cuDNN (recommended if you plan on running Keras on GPU).
- HDF5 and h5py (required if you plan on saving Keras models to disk).
- graphviz and pydot (used by visualization utilities to plot model graphs).

Then, you can install Keras itself. There are two ways to install Keras:

- **Install Keras from PyPI (recommended):**

```
sudo pip install keras
```

If you are using a virtualenv, you may want to avoid using sudo:

```
pip install keras
```

- **Alternatively: install Keras from the GitHub source:**

First, clone Keras using `git`:

```
git clone https://github.com/keras-team/keras.git
```

Then, `cd` to the Keras folder and run the install command:

```
cd keras
sudo python setup.py install
```

Using a different backend than TensorFlow

By default, Keras will use TensorFlow as its tensor manipulation library. Follow these instructions to configure the Keras backend.

Support

You can ask questions and join the development discussion:

- On the Keras Google group.
- On the Keras Slack channel. Use this link to request an invitation to the channel.

You can also post **bug reports and feature requests** (only) in GitHub issues first.

Why this name, Keras?

Keras (   ) means *horn* in Greek. It is a reference to a literary image from ancient Greek and Latin literature, first found in the *Odyssey*, where dream spirits (*Oneiroi*, singular *Oneiros*) are divided between those who deceive men with false visions, who arrive to Earth through a gate of ivory, and those who announce a future that will come to pass, who arrive through a gate of horn. It's a play on the words (horn) / (fulfill), and (ivory) / (deceive).

Keras was initially developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System).

> *"Oneiroi are beyond our unravelling –who can be sure what tale they tell? Not all that men look for comes to pass. Two gates there are that give passage to fleeting Oneiroi; one is made of horn, one of ivory. The Oneiroi that pass through sawn ivory are deceitful, bearing a message that will not be fulfilled; those that come out through polished horn have truth behind them, to be accomplished for men who see them."* Homer, Odyssey 19. 562 ff (Shewring translation).

Why use Keras

Why use Keras?

There are countless deep learning frameworks available today. Why use Keras rather than any other? Here are some of the areas in which Keras compares favorably to existing alternatives.

Keras prioritizes developer experience

- Keras is an API designed for human beings, not machines. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.
- This makes Keras easy to learn and easy to use. As a Keras user, you are more productive, allowing you to try more ideas than your competition, faster – which in turn helps you win machine learning competitions.
- This ease of use does not come at the cost of reduced flexibility: because Keras integrates with lower-level deep learning languages (in particular TensorFlow), it enables you to implement anything you could have built in the base language. In particular, as `tf.keras`, the Keras API integrates seamlessly with your TensorFlow workflows.

Keras has broad adoption in the industry and the research community

With over 200,000 individual users as of November 2017, Keras has stronger adoption in both the industry and the research community than any other deep learning framework except TensorFlow itself (and Keras is commonly used in conjunction with TensorFlow).

You are already constantly interacting with features built with Keras – it is in use at Netflix, Uber, Yelp, Instacart, Zocdoc, Square, and many others. It is especially popular among startups that place deep learning at the core of their products.

Keras is also a favorite among deep learning researchers, coming in #2 in terms of mentions in scientific papers uploaded to the preprint server arXiv.org:

Keras has also been adopted by researchers at large scientific organizations, in particular CERN and NASA.

Keras makes it easy to turn models into products

Your Keras models can be easily deployed across a greater range of platforms than any other deep learning framework:

- On iOS, via Apple's CoreML (Keras support officially provided by Apple). Here's a tutorial.
- On Android, via the TensorFlow Android runtime. Example: Not Hotdog app.
- In the browser, via GPU-accelerated JavaScript runtimes such as Keras.js and WebDNN.
- On Google Cloud, via TensorFlow-Serving.
- In a Python webapp backend (such as a Flask app).
- On the JVM, via DL4J model import provided by SkyMind.
- On Raspberry Pi.

Keras supports multiple backend engines and does not lock you into one ecosystem

Your Keras models can be developed with a range of different deep learning backends. Importantly, any Keras model that only leverages built-in layers will be portable across all these backends: you can train a model with one backend, and load it with another (e.g. for deployment). Available backends include:

- The TensorFlow backend (from Google)
- The CNTK backend (from Microsoft)
- The Theano backend

Amazon is also currently working on developing a MXNet backend for Keras.

As such, your Keras model can be trained on a number of different hardware platforms beyond CPUs:

- NVIDIA GPUs
- Google TPUs, via the TensorFlow backend and Google Cloud
- OpenCL-enabled GPUs, such as those from AMD, via the PlaidML Keras backend

---

Keras has strong multi-GPU support and distributed training support

- Keras has built-in support for multi-GPU data parallelism
- Horovod, from Uber, has first-class support for Keras models
- Keras models can be turned into TensorFlow Estimators and trained on clusters of GPUs on Google Cloud
- Keras can be run on Spark via Dist-Keras (from CERN) and Elephas

---

Keras development is backed by key companies in the deep learning ecosystem

Keras development is backed primarily by Google, and the Keras API comes packaged in TensorFlow as `tf.keras`. Additionally, Microsoft maintains the CNTK Keras backend. Amazon AWS is developing MXNet support. Other contributing companies include NVIDIA, Uber, and Apple (with CoreML).

Guide to the Sequential model

Getting started with the Keras Sequential model

The `Sequential` model is a linear stack of layers.

You can create a `Sequential` model by passing a list of layer instances to the constructor:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

You can also simply add layers via the `.add()` method:

```
model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

Specifying the input shape

The model needs to know what input shape it should expect. For this reason, the first layer in a `Sequential` model (and only the first, because following layers can do automatic shape inference) needs to receive information about its input shape. There are several possible ways to do this:

- Pass an `input_shape` argument to the first layer. This is a shape tuple (a tuple of integers or `None` entries, where `None` indicates that any positive integer may be expected). In `input_shape`, the batch dimension is not included.
- Some 2D layers, such as `Dense`, support the specification of their input shape via the argument `input_dim`, and some 3D temporal layers support the arguments `input_dim` and `input_length`.
- If you ever need to specify a fixed batch size for your inputs (this is useful for stateful recurrent networks), you can pass a `batch_size` argument to a layer. If you pass both `batch_size=32` and `input_shape=(6, 8)` to a layer, it will then expect every batch of inputs to have the batch shape `(32, 6, 8)`.

As such, the following snippets are strictly equivalent:

```
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
```

```
model = Sequential()
model.add(Dense(32, input_dim=784))
```

Compilation

Before training a model, you need to configure the learning process, which is done via the `compile` method. It receives three arguments:

- An optimizer. This could be the string identifier of an existing optimizer (such as `rmsprop` or `adagrad`), or an instance of the `Optimizer` class. See: optimizers.
- A loss function. This is the objective that the model will try to minimize. It can be the string identifier of an existing loss function (such as

categorical_crossentropy or mse), or it can be an objective function. See: losses.
- A list of metrics. For any classification problem you will want to set this to metrics=['accuracy']. A metric could be the string identifier of an existing metric or a custom metric function.

```
### For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

### For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

### For a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')

### For custom metrics
import keras.backend as K

def mean_pred(y_true, y_pred):
    return K.mean(y_pred)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy', mean_pred])
```

Training

Keras models are trained on Numpy arrays of input data and labels. For training a model, you will typically use the fit function. Read its documentation here.

```
### For a single-input model with 2 classes (binary classification):

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

### Generate dummy data
```

```
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))

### Train the model, iterating on the data in batches of 32 samples
model.fit(data, labels, epochs=10, batch_size=32)
```

```
### For a single-input model with 10 classes (categorical classification):

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

### Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))

### Convert labels to categorical one-hot encoding
one_hot_labels = keras.utils.to_categorical(labels, num_classes=10)

### Train the model, iterating on the data in batches of 32 samples
model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

Examples

Here are a few examples to get you started!

In the examples folder, you will also find example models for real datasets:

- CIFAR10 small images classification: Convolutional Neural Network (CNN) with realtime data augmentation
- IMDB movie review sentiment classification: LSTM over sequences of words
- Reuters newswires topic classification: Multilayer Perceptron (MLP)
- MNIST handwritten digits classification: MLP & CNN
- Character-level text generation with LSTM

...and more.

Multilayer Perceptron (MLP) for multi-class softmax classification:

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

### Generate dummy data
import numpy as np
x_train = np.random.random((1000, 20))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)), num_classes=10)
x_test = np.random.random((100, 20))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)

model = Sequential()
### Dense(64) is a fully-connected layer with 64 hidden units.
### in the first layer, you must specify the expected input data shape:
### here, 20-dimensional vectors.
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

MLP for binary classification:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout

### Generate dummy data
x_train = np.random.random((1000, 20))
y_train = np.random.randint(2, size=(1000, 1))
x_test = np.random.random((100, 20))
y_test = np.random.randint(2, size=(100, 1))

model = Sequential()
```

```
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

VGG-like convnet:

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import SGD

### Generate dummy data
x_train = np.random.random((100, 100, 100, 3))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)
x_test = np.random.random((20, 100, 100, 3))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(20, 1)), num_classes=10)

model = Sequential()
### input: 100x100 images with 3 channels -> (100, 100, 3) tensors.
### this applies 32 convolution filters of size 3x3 each.
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(100, 100, 3)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

model.fit(x_train, y_train, batch_size=32, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=32)
```

Sequence classification with LSTM:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Embedding
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, output_dim=256))
model.add(LSTM(128))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train, batch_size=16, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=16)
```

Sequence classification with 1D convolutions:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalAveragePooling1D, MaxPooling1D

model = Sequential()
model.add(Conv1D(64, 3, activation='relu', input_shape=(seq_length, 100)))
model.add(Conv1D(64, 3, activation='relu'))
model.add(MaxPooling1D(3))
model.add(Conv1D(128, 3, activation='relu'))
model.add(Conv1D(128, 3, activation='relu'))
model.add(GlobalAveragePooling1D())
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train, batch_size=16, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=16)
```

Stacked LSTM for sequence classification

In this model, we stack 3 LSTM layers on top of each other, making the model capable of learning higher-level temporal representations.

The first two LSTMs return their full output sequences, but the last one only returns the last step in its output sequence, thus dropping the temporal dimension (i.e. converting the input sequence into a single vector).

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
num_classes = 10

### expected input data shape: (batch_size, timesteps, data_dim)
model = Sequential()
model.add(LSTM(32, return_sequences=True,
               input_shape=(timesteps, data_dim)))  # returns a sequence of vectors of dimen
model.add(LSTM(32, return_sequences=True))  # returns a sequence of vectors of dimension 32
model.add(LSTM(32))  # return a single vector of dimension 32
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

### Generate dummy training data
x_train = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, num_classes))

### Generate dummy validation data
x_val = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, num_classes))

model.fit(x_train, y_train,
          batch_size=64, epochs=5,
          validation_data=(x_val, y_val))
```

Same stacked LSTM model, rendered "stateful"

A stateful recurrent model is one for which the internal states (memories) obtained after processing a batch of samples are reused as initial states for the samples of the next batch. This allows to process longer sequences while keeping computational complexity manageable.

You can read more about stateful RNNs in the FAQ.

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
num_classes = 10
batch_size = 32

### Expected input batch shape: (batch_size, timesteps, data_dim)
### Note that we have to provide the full batch_input_shape since the network is stateful.
### the sample of index i in batch k is the follow-up for the sample i in batch k-1.
model = Sequential()
model.add(LSTM(32, return_sequences=True, stateful=True,
               batch_input_shape=(batch_size, timesteps, data_dim)))
model.add(LSTM(32, return_sequences=True, stateful=True))
model.add(LSTM(32, stateful=True))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

### Generate dummy training data
x_train = np.random.random((batch_size * 10, timesteps, data_dim))
y_train = np.random.random((batch_size * 10, num_classes))

### Generate dummy validation data
x_val = np.random.random((batch_size * 3, timesteps, data_dim))
y_val = np.random.random((batch_size * 3, num_classes))

model.fit(x_train, y_train,
          batch_size=batch_size, epochs=5, shuffle=False,
          validation_data=(x_val, y_val))
```

Guide to the Functional API

Getting started with the Keras functional API

The Keras functional API is the way to go for defining complex models, such as multi-output models, directed acyclic graphs, or models with shared layers.

This guide assumes that you are already familiar with the `Sequential` model.

Let's start with something simple.

First example: a densely-connected network

The `Sequential` model is probably a better choice to implement such a network, but it helps to start with something really simple.

- A layer instance is callable (on a tensor), and it returns a tensor
- Input tensor(s) and output tensor(s) can then be used to define a `Model`
- Such a model can be trained just like Keras `Sequential` models.

```
from keras.layers import Input, Dense
from keras.models import Model

### This returns a tensor
inputs = Input(shape=(784,))

### a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

### This creates a model that includes
### the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

All models are callable, just like layers

With the functional API, it is easy to reuse trained models: you can treat any model as if it were a layer, by calling it on a tensor. Note that by calling a model you aren't just reusing the *architecture* of the model, you are also reusing its weights.

```
x = Input(shape=(784,))
### This works, and returns the 10-way softmax we defined above.
y = model(x)
```

This can allow, for instance, to quickly create models that can process *sequences* of inputs. You could turn an image classification model into a video classification model, in just one line.

```
from keras.layers import TimeDistributed

### Input tensor for sequences of 20 timesteps,
### each containing a 784-dimensional vector
input_sequences = Input(shape=(20, 784))

### This applies our previous model to every timestep in the input sequences.
### the output of the previous model was a 10-way softmax,
### so the output of the layer below will be a sequence of 20 vectors of size 10.
processed_sequences = TimeDistributed(model)(input_sequences)
```

Multi-input and multi-output models

Here's a good use case for the functional API: models with multiple inputs and outputs. The functional API makes it easy to manipulate a large number of intertwined datastreams.

Let's consider the following model. We seek to predict how many retweets and likes a news headline will receive on Twitter. The main input to the model will be the headline itself, as a sequence of words, but to spice things up, our model will also have an auxiliary input, receiving extra data such as the time of day when the headline was posted, etc. The model will also be supervised via two loss functions. Using the main loss function earlier in a model is a good regularization mechanism for deep models.

Here's what our model looks like:

Let's implement it with the functional API.

The main input will receive the headline, as a sequence of integers (each integer encodes a word). The integers will be between 1 and 10,000 (a vocabulary of 10,000 words) and the sequences will be 100 words long.

```
from keras.layers import Input, Embedding, LSTM, Dense
from keras.models import Model

### Headline input: meant to receive sequences of 100 integers, between 1 and 10000.
### Note that we can name any layer by passing it a "name" argument.
main_input = Input(shape=(100,), dtype='int32', name='main_input')

### This embedding layer will encode the input sequence
### into a sequence of dense 512-dimensional vectors.
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)

### A LSTM will transform the vector sequence into a single vector,
### containing information about the entire sequence
lstm_out = LSTM(32)(x)
```

Here we insert the auxiliary loss, allowing the LSTM and Embedding layer to be trained smoothly even though the main loss will be much higher in the model.

```
auxiliary_output = Dense(1, activation='sigmoid', name='aux_output')(lstm_out)
```

At this point, we feed into the model our auxiliary input data by concatenating it with the LSTM output:

```
auxiliary_input = Input(shape=(5,), name='aux_input')
x = keras.layers.concatenate([lstm_out, auxiliary_input])

### We stack a deep densely-connected network on top
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)

### And finally we add the main logistic regression layer
main_output = Dense(1, activation='sigmoid', name='main_output')(x)
```

This defines a model with two inputs and two outputs:

```
model = Model(inputs=[main_input, auxiliary_input], outputs=[main_output, auxiliary_output]
```

We compile the model and assign a weight of 0.2 to the auxiliary loss. To specify different `loss_weights` or `loss` for each different output, you can use a list or a dictionary. Here we pass a single loss as the `loss` argument, so the same loss will be used on all outputs.

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy',
              loss_weights=[1., 0.2])
```

We can train the model by passing it lists of input arrays and target arrays:

```
model.fit([headline_data, additional_data], [labels, labels],
          epochs=50, batch_size=32)
```

Since our inputs and outputs are named (we passed them a "name" argument), we could also have compiled the model via:

```
model.compile(optimizer='rmsprop',
              loss={'main_output': 'binary_crossentropy', 'aux_output': 'binary_crossentrop
              loss_weights={'main_output': 1., 'aux_output': 0.2})

### And trained it via:
model.fit({'main_input': headline_data, 'aux_input': additional_data},
          {'main_output': labels, 'aux_output': labels},
          epochs=50, batch_size=32)
```

Shared layers

Another good use for the functional API are models that use shared layers. Let's take a look at shared layers.

Let's consider a dataset of tweets. We want to build a model that can tell whether two tweets are from the same person or not (this can allow us to compare users by the similarity of their tweets, for instance).

One way to achieve this is to build a model that encodes two tweets into two vectors, concatenates the vectors and then adds a logistic regression; this outputs a probability that the two tweets share the same author. The model would then be trained on positive tweet pairs and negative tweet pairs.

Because the problem is symmetric, the mechanism that encodes the first tweet should be reused (weights and all) to encode the second tweet. Here we use a shared LSTM layer to encode the tweets.

Let's build this with the functional API. We will take as input for a tweet a binary matrix of shape `(280, 256)`, i.e. a sequence of 280 vectors of size 256, where each dimension in the 256-dimensional vector encodes the presence/absence of a character (out of an alphabet of 256 frequent characters).

```
import keras
from keras.layers import Input, LSTM, Dense
from keras.models import Model

tweet_a = Input(shape=(280, 256))
tweet_b = Input(shape=(280, 256))
```

To share a layer across different inputs, simply instantiate the layer once, then call it on as many inputs as you want:

```
### This layer can take as input a matrix
### and will return a vector of size 64
shared_lstm = LSTM(64)

### When we reuse the same layer instance
### multiple times, the weights of the layer
### are also being reused
### (it is effectively *the same* layer)
encoded_a = shared_lstm(tweet_a)
encoded_b = shared_lstm(tweet_b)

### We can then concatenate the two vectors:
merged_vector = keras.layers.concatenate([encoded_a, encoded_b], axis=-1)

### And add a logistic regression on top
predictions = Dense(1, activation='sigmoid')(merged_vector)

### We define a trainable model linking the
### tweet inputs to the predictions
model = Model(inputs=[tweet_a, tweet_b], outputs=predictions)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.fit([data_a, data_b], labels, epochs=10)
```

Let's pause to take a look at how to read the shared layer's output or output shape.

The concept of layer "node"

Whenever you are calling a layer on some input, you are creating a new tensor (the output of the layer), and you are adding a "node" to the layer, linking the input tensor to the output tensor. When you are calling the same layer multiple times, that layer owns multiple nodes indexed as 0, 1, 2…

In previous versions of Keras, you could obtain the output tensor of a layer instance via `layer.get_output()`, or its output shape via `layer.output_shape`. You still can (except `get_output()` has been replaced by the property `output`). But what if a layer is connected to multiple inputs?

As long as a layer is only connected to one input, there is no confusion, and `.output` will return the one output of the layer:

```
a = Input(shape=(280, 256))

lstm = LSTM(32)
encoded_a = lstm(a)

assert lstm.output == encoded_a
```

Not so if the layer has multiple inputs:

```
a = Input(shape=(280, 256))
b = Input(shape=(280, 256))

lstm = LSTM(32)
encoded_a = lstm(a)
encoded_b = lstm(b)

lstm.output
```

```
>> AttributeError: Layer lstm_1 has multiple inbound nodes,
hence the notion of "layer output" is ill-defined.
Use `get_output_at(node_index)` instead.
```

Okay then. The following works:

```
assert lstm.get_output_at(0) == encoded_a
assert lstm.get_output_at(1) == encoded_b
```

Simple enough, right?

The same is true for the properties `input_shape` and `output_shape`: as long as the layer has only one node, or as long as all nodes have the same input/output shape, then the notion of "layer output/input shape" is well defined, and that one shape will be returned by `layer.output_shape`/`layer.input_shape`. But if, for instance, you apply the same `Conv2D` layer to an input of shape `(32, 32, 3)`, and then to an input of shape `(64, 64, 3)`, the layer will have multiple input/output shapes, and you will have to fetch them by specifying the index of the node they belong to:

```
a = Input(shape=(32, 32, 3))
b = Input(shape=(64, 64, 3))

conv = Conv2D(16, (3, 3), padding='same')
conved_a = conv(a)

### Only one input so far, the following will work:
assert conv.input_shape == (None, 32, 32, 3)

conved_b = conv(b)
### now the `.input_shape` property wouldn't work, but this does:
assert conv.get_input_shape_at(0) == (None, 32, 32, 3)
assert conv.get_input_shape_at(1) == (None, 64, 64, 3)
```

More examples

Code examples are still the best way to get started, so here are a few more.

Inception module

For more information about the Inception architecture, see Going Deeper with Convolutions.

```
from keras.layers import Conv2D, MaxPooling2D, Input

input_img = Input(shape=(256, 256, 3))

tower_1 = Conv2D(64, (1, 1), padding='same', activation='relu')(input_img)
tower_1 = Conv2D(64, (3, 3), padding='same', activation='relu')(tower_1)

tower_2 = Conv2D(64, (1, 1), padding='same', activation='relu')(input_img)
tower_2 = Conv2D(64, (5, 5), padding='same', activation='relu')(tower_2)

tower_3 = MaxPooling2D((3, 3), strides=(1, 1), padding='same')(input_img)
tower_3 = Conv2D(64, (1, 1), padding='same', activation='relu')(tower_3)

output = keras.layers.concatenate([tower_1, tower_2, tower_3], axis=1)
```

Residual connection on a convolution layer

For more information about residual networks, see Deep Residual Learning for Image Recognition.

```
from keras.layers import Conv2D, Input

### input tensor for a 3-channel 256x256 image
x = Input(shape=(256, 256, 3))
### 3x3 conv with 3 output channels (same as input channels)
y = Conv2D(3, (3, 3), padding='same')(x)
### this returns x + y.
z = keras.layers.add([x, y])
```

Shared vision model

This model reuses the same image-processing module on two inputs, to classify whether two MNIST digits are the same digit or different digits.

```
from keras.layers import Conv2D, MaxPooling2D, Input, Dense, Flatten
from keras.models import Model

### First, define the vision modules
digit_input = Input(shape=(27, 27, 1))
x = Conv2D(64, (3, 3))(digit_input)
x = Conv2D(64, (3, 3))(x)
x = MaxPooling2D((2, 2))(x)
```

```
out = Flatten()(x)

vision_model = Model(digit_input, out)

### Then define the tell-digits-apart model
digit_a = Input(shape=(27, 27, 1))
digit_b = Input(shape=(27, 27, 1))

### The vision model will be shared, weights and all
out_a = vision_model(digit_a)
out_b = vision_model(digit_b)

concatenated = keras.layers.concatenate([out_a, out_b])
out = Dense(1, activation='sigmoid')(concatenated)

classification_model = Model([digit_a, digit_b], out)
```

Visual question answering model

This model can select the correct one-word answer when asked a natural-language question about a picture.

It works by encoding the question into a vector, encoding the image into a vector, concatenating the two, and training on top a logistic regression over some vocabulary of potential answers.

```
from keras.layers import Conv2D, MaxPooling2D, Flatten
from keras.layers import Input, LSTM, Embedding, Dense
from keras.models import Model, Sequential

### First, let's define a vision model using a Sequential model.
### This model will encode an image into a vector.
vision_model = Sequential()
vision_model.add(Conv2D(64, (3, 3), activation='relu', padding='same', input_shape=(224, 22
vision_model.add(Conv2D(64, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
vision_model.add(Conv2D(128, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
vision_model.add(Conv2D(256, (3, 3), activation='relu'))
vision_model.add(Conv2D(256, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Flatten())

### Now let's get a tensor with the output of our vision model:
```

```
image_input = Input(shape=(224, 224, 3))
encoded_image = vision_model(image_input)

### Next, let's define a language model to encode the question into a vector.
### Each question will be at most 100 word long,
### and we will index words as integers from 1 to 9999.
question_input = Input(shape=(100,), dtype='int32')
embedded_question = Embedding(input_dim=10000, output_dim=256, input_length=100)(question_in
encoded_question = LSTM(256)(embedded_question)

### Let's concatenate the question vector and the image vector:
merged = keras.layers.concatenate([encoded_question, encoded_image])

### And let's train a logistic regression over 1000 words on top:
output = Dense(1000, activation='softmax')(merged)

### This is our final model:
vqa_model = Model(inputs=[image_input, question_input], outputs=output)

### The next stage would be training this model on actual data.
```

Video question answering model

Now that we have trained our image QA model, we can quickly turn it into a video QA model. With appropriate training, you will be able to show it a short video (e.g. 100-frame human action) and ask a natural language question about the video (e.g. "what sport is the boy playing?" -> "football").

```
from keras.layers import TimeDistributed

video_input = Input(shape=(100, 224, 224, 3))
### This is our video encoded via the previously trained vision_model (weights are reused)
encoded_frame_sequence = TimeDistributed(vision_model)(video_input)  # the output will be a
encoded_video = LSTM(256)(encoded_frame_sequence)  # the output will be a vector

### This is a model-level representation of the question encoder, reusing the same weights a
question_encoder = Model(inputs=question_input, outputs=encoded_question)

### Let's use it to encode the question:
video_question_input = Input(shape=(100,), dtype='int32')
encoded_video_question = question_encoder(video_question_input)

### And this is our video question answering model:
merged = keras.layers.concatenate([encoded_video, encoded_video_question])
output = Dense(1000, activation='softmax')(merged)
video_qa_model = Model(inputs=[video_input, video_question_input], outputs=output)
```

FAQ

Keras FAQ: Frequently Asked Keras Questions

- How should I cite Keras?
- How can I run Keras on GPU?
- How can I run a Keras model on multiple GPUs?
- What does "sample", "batch", "epoch" mean?
- How can I save a Keras model?
- Why is the training loss much higher than the testing loss?
- How can I obtain the output of an intermediate layer?
- How can I use Keras with datasets that don't fit in memory?
- How can I interrupt training when the validation loss isn't decreasing anymore?
- How is the validation split computed?
- Is the data shuffled during training?
- How can I record the training / validation loss / accuracy at each epoch?
- How can I "freeze" layers?
- How can I use stateful RNNs?
- How can I remove a layer from a Sequential model?
- How can I use pre-trained models in Keras?
- How can I use HDF5 inputs with Keras?
- Where is the Keras configuration file stored?
- How can I obtain reproducible results using Keras during development?
- How can I install HDF5 or h5py to save my models in Keras?

How should I cite Keras?

Please cite Keras in your publications if it helps your research. Here is an example BibTeX entry:

```
@misc{chollet2015keras,
  title={Keras},
  author={Chollet, Fran\c{c}ois and others},
  year={2015},
  howpublished={\url{https://keras.io}},
}
```

How can I run Keras on GPU?

If you are running on the **TensorFlow** or **CNTK** backends, your code will automatically run on GPU if any available GPU is detected.

If you are running on the **Theano** backend, you can use one of the following methods:

Method 1: use Theano flags.

```
THEANO_FLAGS=device=gpu,floatX=float32 python my_keras_script.py
```

The name 'gpu' might have to be changed depending on your device's identifier (e.g. `gpu0`, `gpu1`, etc).

Method 2: set up your `.theanorc`: Instructions

Method 3: manually set `theano.config.device`, `theano.config.floatX` at the beginning of your code:

```
import theano
theano.config.device = 'gpu'
theano.config.floatX = 'float32'
```

How can I run a Keras model on multiple GPUs?

We recommend doing so using the **TensorFlow** backend. There are two ways to run a single model on multiple GPUs: **data parallelism** and **device parallelism**.

In most cases, what you need is most likely data parallelism.

Data parallelism

Data parallelism consists in replicating the target model once on each device, and using each replica to process a different fraction of the input data. Keras has a built-in utility, `keras.utils.multi_gpu_model`, which can produce a data-parallel version of any model, and achieves quasi-linear speedup on up to 8 GPUs.

For more information, see the documentation for multi_gpu_model. Here is a quick example:

```
from keras.utils import multi_gpu_model

### Replicates `model` on 8 GPUs.
### This assumes that your machine has 8 available GPUs.
parallel_model = multi_gpu_model(model, gpus=8)
parallel_model.compile(loss='categorical_crossentropy',
                       optimizer='rmsprop')

### This `fit` call will be distributed on 8 GPUs.
### Since the batch size is 256, each GPU will process 32 samples.
parallel_model.fit(x, y, epochs=20, batch_size=256)
```

Device parallelism

Device parallelism consists in running different parts of a same model on different devices. It works best for models that have a parallel architecture, e.g. a model with two branches.

This can be achieved by using TensorFlow device scopes. Here is a quick example:

```
### Model where a shared LSTM is used to encode two different sequences in parallel
input_a = keras.Input(shape=(140, 256))
input_b = keras.Input(shape=(140, 256))

shared_lstm = keras.layers.LSTM(64)

### Process the first sequence on one GPU
with tf.device_scope('/gpu:0'):
    encoded_a = shared_lstm(tweet_a)
### Process the next sequence on another GPU
with tf.device_scope('/gpu:1'):
    encoded_b = shared_lstm(tweet_b)

### Concatenate results on CPU
with tf.device_scope('/cpu:0'):
    merged_vector = keras.layers.concatenate([encoded_a, encoded_b],
                                             axis=-1)
```

What does “sample”, “batch”, “epoch” mean?

Below are some common definitions that are necessary to know and understand to correctly utilize Keras:

- **Sample**: one element of a dataset.
- *Example:* one image is a **sample** in a convolutional network
- *Example:* one audio file is a **sample** for a speech recognition model
- **Batch**: a set of N samples. The samples in a **batch** are processed independently, in parallel. If training, a batch results in only one update to the model.
- A **batch** generally approximates the distribution of the input data better than a single input. The larger the batch, the better the approximation; however, it is also true that the batch will take longer to process and will still result in only one update. For inference (evaluate/predict), it is recommended to pick a batch size that is as large as you can afford without going out of memory (since larger batches will usually result in faster evaluating/prediction).

- **Epoch**: an arbitrary cutoff, generally defined as "one pass over the entire dataset", used to separate training into distinct phases, which is useful for logging and periodic evaluation.
- When using `evaluation_data` or `evaluation_split` with the `fit` method of Keras models, evaluation will be run at the end of every **epoch**.
- Within Keras, there is the ability to add callbacks specifically designed to be run at the end of an **epoch**. Examples of these are learning rate changes and model checkpointing (saving).

How can I save a Keras model?

Saving/loading whole models (architecture + weights + optimizer state)

It is not recommended to use pickle or cPickle to save a Keras model.

You can use `model.save(filepath)` to save a Keras model into a single HDF5 file which will contain:

- the architecture of the model, allowing to re-create the model
- the weights of the model
- the training configuration (loss, optimizer)
- the state of the optimizer, allowing to resume training exactly where you left off.

You can then use `keras.models.load_model(filepath)` to reinstantiate your model. `load_model` will also take care of compiling the model using the saved training configuration (unless the model was never compiled in the first place).

Please also see How can I install HDF5 or h5py to save my models in Keras? for instructions on how to install `h5py`.

Example:

```
from keras.models import load_model

model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
del model  # deletes the existing model

### returns a compiled model
### identical to the previous one
model = load_model('my_model.h5')
```

Saving/loading only a model's architecture

If you only need to save the **architecture of a model**, and not its weights or its training configuration, you can do:

```
### save as JSON
json_string = model.to_json()

### save as YAML
yaml_string = model.to_yaml()
```

The generated JSON / YAML files are human-readable and can be manually edited if needed.

You can then build a fresh model from this data:

```
### model reconstruction from JSON:
from keras.models import model_from_json
model = model_from_json(json_string)

### model reconstruction from YAML
from keras.models import model_from_yaml
model = model_from_yaml(yaml_string)
```

Saving/loading only a model's weights

If you need to save the **weights of a model**, you can do so in HDF5 with the code below.

```
model.save_weights('my_model_weights.h5')
```

Assuming you have code for instantiating your model, you can then load the weights you saved into a model with the *same* architecture:

```
model.load_weights('my_model_weights.h5')
```

If you need to load weights into a *different* architecture (with some layers in common), for instance for fine-tuning or transfer-learning, you can load weights by *layer name*:

```
model.load_weights('my_model_weights.h5', by_name=True)
```

Please also see How can I install HDF5 or h5py to save my models in Keras? for instructions on how to install `h5py`.

For example:

```
"""
Assuming the original model looks like this:
    model = Sequential()
    model.add(Dense(2, input_dim=3, name='dense_1'))
    model.add(Dense(3, name='dense_2'))
    ...
    model.save_weights(fname)
"""

### new model
```

```
model = Sequential()
model.add(Dense(2, input_dim=3, name='dense_1'))  # will be loaded
model.add(Dense(10, name='new_dense'))  # will not be loaded

### load weights from first model; will only affect the first layer, dense_1.
model.load_weights(fname, by_name=True)
```

Handling custom layers (or other custom objects) in saved models

If the model you want to load includes custom layers or other custom classes or functions, you can pass them to the loading mechanism via the `custom_objects` argument:

```
from keras.models import load_model
### Assuming your model includes instance of an "AttentionLayer" class
model = load_model('my_model.h5', custom_objects={'AttentionLayer': AttentionLayer})
```

Alternatively, you can use a custom object scope:

```
from keras.utils import CustomObjectScope

with CustomObjectScope({'AttentionLayer': AttentionLayer}):
    model = load_model('my_model.h5')
```

Custom objects handling works the same way for `load_model`, `model_from_json`, `model_from_yaml`:

```
from keras.models import model_from_json
model = model_from_json(json_string, custom_objects={'AttentionLayer': AttentionLayer})
```

Why is the training loss much higher than the testing loss?

A Keras model has two modes: training and testing. Regularization mechanisms, such as Dropout and L1/L2 weight regularization, are turned off at testing time.

Besides, the training loss is the average of the losses over each batch of training data. Because your model is changing over time, the loss over the first batches of an epoch is generally higher than over the last batches. On the other hand, the testing loss for an epoch is computed using the model as it is at the end of the epoch, resulting in a lower loss.

How can I obtain the output of an intermediate layer?

One simple way is to create a new `Model` that will output the layers that you are interested in:

```
from keras.models import Model

model = ...  # create the original model

layer_name = 'my_layer'
intermediate_layer_model = Model(inputs=model.input,
                                 outputs=model.get_layer(layer_name).output)
intermediate_output = intermediate_layer_model.predict(data)
```

Alternatively, you can build a Keras function that will return the output of a certain layer given a certain input, for example:

```
from keras import backend as K

### with a Sequential model
get_3rd_layer_output = K.function([model.layers[0].input],
                                  [model.layers[3].output])
layer_output = get_3rd_layer_output([x])[0]
```

Similarly, you could build a Theano and TensorFlow function directly.

Note that if your model has a different behavior in training and testing phase (e.g. if it uses `Dropout`, `BatchNormalization`, etc.), you will need to pass the learning phase flag to your function:

```
get_3rd_layer_output = K.function([model.layers[0].input, K.learning_phase()],
                                  [model.layers[3].output])

### output in test mode = 0
layer_output = get_3rd_layer_output([x, 0])[0]

### output in train mode = 1
layer_output = get_3rd_layer_output([x, 1])[0]
```

How can I use Keras with datasets that don't fit in memory?

You can do batch training using `model.train_on_batch(x, y)` and `model.test_on_batch(x, y)`. See the models documentation.

Alternatively, you can write a generator that yields batches of training data and use the method `model.fit_generator(data_generator, steps_per_epoch, epochs)`.

You can see batch training in action in our CIFAR10 example.

How can I interrupt training when the validation loss isn't decreasing anymore?

You can use an `EarlyStopping` callback:

```
from keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(monitor='val_loss', patience=2)
model.fit(x, y, validation_split=0.2, callbacks=[early_stopping])
```

Find out more in the callbacks documentation.

How is the validation split computed?

If you set the `validation_split` argument in `model.fit` to e.g. 0.1, then the validation data used will be the *last 10%* of the data. If you set it to 0.25, it will be the last 25% of the data, etc. Note that the data isn't shuffled before extracting the validation split, so the validation is literally just the *last* x% of samples in the input you passed.

The same validation set is used for all epochs (within a same call to `fit`).

Is the data shuffled during training?

Yes, if the `shuffle` argument in `model.fit` is set to `True` (which is the default), the training data will be randomly shuffled at each epoch.

Validation data is never shuffled.

How can I record the training / validation loss / accuracy at each epoch?

The `model.fit` method returns an `History` callback, which has a `history` attribute containing the lists of successive losses and other metrics.

```
hist = model.fit(x, y, validation_split=0.2)
print(hist.history)
```

How can I "freeze" Keras layers?

To "freeze" a layer means to exclude it from training, i.e. its weights will never be updated. This is useful in the context of fine-tuning a model, or using fixed embeddings for a text input.

You can pass a `trainable` argument (boolean) to a layer constructor to set a layer to be non-trainable:

```
frozen_layer = Dense(32, trainable=False)
```

Additionally, you can set the `trainable` property of a layer to `True` or `False` after instantiation. For this to take effect, you will need to call `compile()` on your model after modifying the `trainable` property. Here's an example:

```
x = Input(shape=(32,))
layer = Dense(32)
layer.trainable = False
y = layer(x)

frozen_model = Model(x, y)
### in the model below, the weights of `layer` will not be updated during training
frozen_model.compile(optimizer='rmsprop', loss='mse')

layer.trainable = True
trainable_model = Model(x, y)
### with this model the weights of the layer will be updated during training
### (which will also affect the above model since it uses the same layer instance)
trainable_model.compile(optimizer='rmsprop', loss='mse')

frozen_model.fit(data, labels)  # this does NOT update the weights of `layer`
trainable_model.fit(data, labels)  # this updates the weights of `layer`
```

How can I use stateful RNNs?

Making a RNN stateful means that the states for the samples of each batch will be reused as initial states for the samples in the next batch.

When using stateful RNNs, it is therefore assumed that:

- all batches have the same number of samples
- If `x1` and `x2` are successive batches of samples, then `x2[i]` is the follow-up sequence to `x1[i]`, for every `i`.

To use statefulness in RNNs, you need to:

- explicitly specify the batch size you are using, by passing a `batch_size` argument to the first layer in your model. E.g. `batch_size=32` for a 32-samples batch of sequences of 10 timesteps with 16 features per timestep.
- set `stateful=True` in your RNN layer(s).
- specify `shuffle=False` when calling fit().

To reset the states accumulated:

- use `model.reset_states()` to reset the states of all layers in the model
- use `layer.reset_states()` to reset the states of a specific stateful RNN layer

Example:

```
x  # this is our input data, of shape (32, 21, 16)
### we will feed it to our model in sequences of length 10

model = Sequential()
model.add(LSTM(32, input_shape=(10, 16), batch_size=32, stateful=True))
model.add(Dense(16, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

### we train the network to predict the 11th timestep given the first 10:
model.train_on_batch(x[:, :10, :], np.reshape(x[:, 10, :], (32, 16)))

### the state of the network has changed. We can feed the follow-up sequences:
model.train_on_batch(x[:, 10:20, :], np.reshape(x[:, 20, :], (32, 16)))

### let's reset the states of the LSTM layer:
model.reset_states()

### another way to do it in this case:
model.layers[0].reset_states()
```

Notes that the methods `predict`, `fit`, `train_on_batch`, `predict_classes`, etc. will *all* update the states of the stateful layers in a model. This allows you to do not only stateful training, but also stateful prediction.

How can I remove a layer from a Sequential model?

You can remove the last added layer in a Sequential model by calling `.pop()`:

```
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=784))
model.add(Dense(32, activation='relu'))

print(len(model.layers))  # "2"

model.pop()
print(len(model.layers))  # "1"
```

How can I use pre-trained models in Keras?

Code and pre-trained weights are available for the following image classification models:

- Xception
- VGG16
- VGG19
- ResNet50
- Inception v3
- Inception-ResNet v2
- MobileNet v1

They can be imported from the module `keras.applications`:

```
from keras.applications.xception import Xception
from keras.applications.vgg16 import VGG16
from keras.applications.vgg19 import VGG19
from keras.applications.resnet50 import ResNet50
from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_resnet_v2 import InceptionResNetV2
from keras.applications.mobilenet import MobileNet

model = VGG16(weights='imagenet', include_top=True)
```

For a few simple usage examples, see the documentation for the Applications module.

For a detailed example of how to use such a pre-trained model for feature extraction or for fine-tuning, see this blog post.

The VGG16 model is also the basis for several Keras example scripts:

- Style transfer
- Feature visualization
- Deep dream

How can I use HDF5 inputs with Keras?

You can use the `HDF5Matrix` class from `keras.utils.io_utils`. See the HDF5Matrix documentation for details.

You can also directly use a HDF5 dataset:

```
import h5py
with h5py.File('input/file.hdf5', 'r') as f:
    x_data = f['x_data']
    model.predict(x_data)
```

Please also see How can I install HDF5 or h5py to save my models in Keras? for instructions on how to install `h5py`.

Where is the Keras configuration file stored?

The default directory where all Keras data is stored is:

`$HOME/.keras/`

Note that Windows users should replace `$HOME` with `%USERPROFILE%`. In case Keras cannot create the above directory (e.g. due to permission issues), `/tmp/.keras/` is used as a backup.

The Keras configuration file is a JSON file stored at `$HOME/.keras/keras.json`. The default configuration file looks like this:

```
{
    "image_data_format": "channels_last",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "tensorflow"
}
```

It contains the following fields:

- The image data format to be used as default by image processing layers and utilities (either `channels_last` or `channels_first`).
- The `epsilon` numerical fuzz factor to be used to prevent division by zero in some operations.
- The default float data type.
- The default backend. See the backend documentation.

Likewise, cached dataset files, such as those downloaded with `get_file()`, are stored by default in `$HOME/.keras/datasets/`.

How can I obtain reproducible results using Keras during development?

During development of a model, sometimes it is useful to be able to obtain reproducible results from run to run in order to determine if a change in performance is due to an actual model or data modification, or merely a result of a new random sample. The below snippet of code provides an example of how to obtain reproducible results - this is geared towards a TensorFlow backend for a Python 3 environment.

```
import numpy as np
import tensorflow as tf
import random as rn

### The below is necessary in Python 3.2.3 onwards to
### have reproducible behavior for certain hash-based operations.
### See these references for further details:
### https://docs.python.org/3.4/using/cmdline.html#envvar-PYTHONHASHSEED
### https://github.com/keras-team/keras/issues/2280#issuecomment-306959926

import os
os.environ['PYTHONHASHSEED'] = '0'

### The below is necessary for starting Numpy generated random numbers
### in a well-defined initial state.

np.random.seed(42)

### The below is necessary for starting core Python generated random numbers
### in a well-defined state.

rn.seed(12345)

### Force TensorFlow to use single thread.
### Multiple threads are a potential source of
### non-reproducible results.
### For further details, see: https://stackoverflow.com/questions/42022950/which-seeds-have-

session_conf = tf.ConfigProto(intra_op_parallelism_threads=1, inter_op_parallelism_threads=

from keras import backend as K

### The below tf.set_random_seed() will make random number generation
### in the TensorFlow backend have a well-defined initial state.
### For further details, see: https://www.tensorflow.org/api_docs/python/tf/set_random_seed

tf.set_random_seed(1234)

sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)
K.set_session(sess)

### Rest of code follows ...
```

How can I install HDF5 or h5py to save my models in Keras?

In order to save your Keras models as HDF5 files, e.g. via `keras.callbacks.ModelCheckpoint`, Keras uses the h5py Python package. It is a dependency of Keras and should be installed by default. On Debian-based distributions, you will have to additionally install `libhdf5`:

```
sudo apt-get install libhdf5-serial-dev
```

If you are unsure if h5py is installed you can open a Python shell and load the module via

```
import h5py
```

If it imports without error it is installed otherwise you can find detailed installation instructions here: http://docs.h5py.org/en/latest/build.html

About Keras models

About Keras models

There are two types of models available in Keras: the Sequential model and the Model class used with functional API.

These models have a number of methods in common:

- `model.summary()`: prints a summary representation of your model. Shortcut for utils.print_summary
- `model.get_config()`: returns a dictionary containing the configuration of the model. The model can be reinstantiated from its config via:

  ```
  config = model.get_config()
  model = Model.from_config(config)
  ### or, for Sequential:
  model = Sequential.from_config(config)
  ```

- `model.get_weights()`: returns a list of all weight tensors in the model, as Numpy arrays.
- `model.set_weights(weights)`: sets the values of the weights of the model, from a list of Numpy arrays. The arrays in the list should have the same shape as those returned by `get_weights()`.
- `model.to_json()`: returns a representation of the model as a JSON string. Note that the representation does not include the weights, only the architecture. You can reinstantiate the same model (with reinitialized weights) from the JSON string via: "'python from keras.models import model_from_json

json_string = model.to_json() model = model_from_json(json_string) `- `model.to_yaml()`: returns a representation of the model as a YAML string. Note that the representation does not include the` `weights, only the architecture. You can reinstantiate the same model (with reinitialized weights) from the YAML string via:`python from keras.models import model_from_yaml

yaml_string = model.to_yaml() model = model_from_yaml(yaml_string) “-model.save_weights(filepath)`: saves the weights of the model as a HDF5 file.` -model.load_weights(filepath, by_name=False)`: loads the weights of the model from a HDF5 file (created by`save_weights`). By default, the architecture is expected to be unchanged. To load weights into a different architecture (with some layers in common), use`by_name=True‘ to load only those layers with the same name.

Note: Please also see How can I install HDF5 or h5py to save my models in Keras? in the FAQ for instructions on how to install `h5py`.

Sequential

The Sequential model API

To get started, read this guide to the Keras Sequential model.

Useful attributes of Model

- `model.layers` is a list of the layers added to the model.

Sequential model methods

compile

```
compile(self, optimizer, loss=None, metrics=None, loss_weights=None, sample_weight_mode=None
```

Configures the model for training.

Arguments

- **optimizer**: String (name of optimizer) or optimizer instance. See optimizers.
- **loss**: String (name of objective function) or objective function. See losses. If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of losses. The loss value that will be minimized by the model will then be the sum of all individual losses.
- **metrics**: List of metrics to be evaluated by the model during training and testing. Typically you will use `metrics=['accuracy']`. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as `metrics={'output_a': 'accuracy'}`.

- **loss__weights**: Optional list or dictionary specifying scalar coefficients (Python floats) to weight the loss contributions of different model outputs. The loss value that will be minimized by the model will then be the *weighted sum* of all individual losses, weighted by the `loss_weights` coefficients. If a list, it is expected to have a 1:1 mapping to the model's outputs. If a tensor, it is expected to map output names (strings) to scalar coefficients.
- **sample__weight__mode**: If you need to do timestep-wise sample weighting (2D weights), set this to `"temporal"`. `None` defaults to sample-wise weights (1D). If the model has multiple outputs, you can use a different `sample_weight_mode` on each output by passing a dictionary or a list of modes.
- **weighted__metrics**: List of metrics to be evaluated and weighted by sample__weight or class__weight during training and testing.
- **target__tensors**: By default, Keras will create placeholders for the model's target, which will be fed with the target data during training. If instead you would like to use your own target tensors (in turn, Keras will not expect external Numpy data for these targets at training time), you can specify them via the `target_tensors` argument. It can be a single tensor (for a single-output model), a list of tensors, or a dict mapping output names to target tensors.
- ___**kwargs___: When using the Theano/CNTK backends, these arguments are passed into `K.function`. When using the TensorFlow backend, these arguments are passed into `tf.Session.run`.

Raises

- **ValueError**: In case of invalid arguments for `optimizer`, `loss`, `metrics` or `sample_weight_mode`.

---

fit

```
fit(self, x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_s
```

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

Arguments

- **x**: Numpy array of training data (if the model has a single input), or list of Numpy arrays (if the model has multiple inputs). If input layers in the model are named, you can also pass a dictionary mapping input names to Numpy arrays. `x` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **y**: Numpy array of target (label) data (if the model has a single output), or list of Numpy arrays (if the model has multiple outputs). If output layers in the model are named, you can also pass a dictionary mapping

output names to Numpy arrays. `y` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).

- **batch_size**: Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32.
- **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided. Note that in conjunction with `initial_epoch`, `epochs` is to be understood as "final epoch". The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.
- **verbose**: Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **validation_split**: Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data is selected from the last samples in the `x` and `y` data provided, before shuffling.
- **validation_data**: tuple `(x_val, y_val)` or tuple `(x_val, y_val, val_sample_weights)` on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. `validation_data` will override `validation_split`.
- **shuffle**: Boolean (whether to shuffle the training data before each epoch) or str (for 'batch'). 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when `steps_per_epoch` is not `None`.
- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class.
- **sample_weight**: Optional Numpy array of weights for the training samples, used for weighting the loss function (during training only). You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape `(samples, sequence_length)`, to apply a different weight to every timestep of every sample. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.
- **initial_epoch**: Integer. Epoch at which to start training (useful for resuming a previous training run).
- **steps_per_epoch**: Integer or `None`. Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. When training with input tensors such as TensorFlow data tensors, the default `None` is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined.

- **validation_steps**: Only relevant if `steps_per_epoch` is specified. Total number of steps (batches of samples) to validate before stopping.

Returns

A `History` object. Its `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

Raises

- **RuntimeError**: If the model was never compiled.
- **ValueError**: In case of mismatch between the provided input data and what the model expects.

---

evaluate

```
evaluate(self, x=None, y=None, batch_size=None, verbose=1, sample_weight=None, steps=None)
```

Returns the loss value & metrics values for the model in test mode.

Computation is done in batches.

Arguments

- **x**: Numpy array of test data (if the model has a single input), or list of Numpy arrays (if the model has multiple inputs). If input layers in the model are named, you can also pass a dictionary mapping input names to Numpy arrays. `x` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **y**: Numpy array of target (label) data (if the model has a single output), or list of Numpy arrays (if the model has multiple outputs). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays. `y` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **batch_size**: Integer or `None`. Number of samples per evaluation step. If unspecified, `batch_size` will default to 32.
- **verbose**: 0 or 1. Verbosity mode. 0 = silent, 1 = progress bar.
- **sample_weight**: Optional Numpy array of weights for the test samples, used for weighting the loss function. You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape `(samples, sequence_length)`, to apply a different weight to every timestep of every sample. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.
- **steps**: Integer or `None`. Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of `None`.

Returns

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

predict

```
predict(self, x, batch_size=None, verbose=0, steps=None)
```

Generates output predictions for the input samples.

Computation is done in batches.

Arguments

- **x**: The input data, as a Numpy array (or list of Numpy arrays if the model has multiple outputs).
- **batch_size**: Integer. If unspecified, it will default to 32.
- **verbose**: Verbosity mode, 0 or 1.
- **steps**: Total number of steps (batches of samples) before declaring the prediction round finished. Ignored with the default value of `None`.

Returns

Numpy array(s) of predictions.

Raises

- **ValueError**: In case of mismatch between the provided input data and the model's expectations, or in case a stateful model receives a number of samples that is not a multiple of the batch size.

---

train_on_batch

```
train_on_batch(self, x, y, sample_weight=None, class_weight=None)
```

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

Arguments

- **x**: Numpy array of training data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.

- **sample_weight**: Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify sample_weight_mode="temporal" in compile().
- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) to apply to the model's loss for the samples from this class during training. This can be useful to tell the model to "pay more attention" to samples from an under-represented class.

Returns

Scalar training loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

test_on_batch

```
test_on_batch(self, x, y, sample_weight=None)
```

Test the model on a single batch of samples.

Arguments

- **x**: Numpy array of test data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **sample_weight**: Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify sample_weight_mode="temporal" in compile().

Returns

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

predict_on_batch

```
predict_on_batch(self, x)
```

Returns predictions for a single batch of samples.

Arguments

- **x**: Input samples, as a Numpy array.

Returns

Numpy array(s) of predictions.

---

fit_generator

```
fit_generator(self, generator, steps_per_epoch=None, epochs=1, verbose=1, callbacks=None, va
```

Trains the model on data generated batch-by-batch by a Python generator (or an instance of `Sequence`).

The generator is run in parallel to the model, for efficiency. For instance, this allows you to do real-time data augmentation on images on CPU in parallel to training your model on GPU.

The use of `keras.utils.Sequence` guarantees the ordering and guarantees the single use of every input per epoch when using `use_multiprocessing=True`.

Arguments

- **generator**: A generator or an instance of `Sequence` (`keras.utils.Sequence`) object in order to avoid duplicate data when using multiprocessing. The output of the generator must be either
- a tuple `(inputs, targets)`
- a tuple `(inputs, targets, sample_weights)`. This tuple (a single output of the generator) makes a single batch. Therefore, all arrays in this tuple must have the same length (equal to the size of this batch). Different batches may have different sizes. For example, the last batch of the epoch is commonly smaller than the others, if the size of the dataset is not divisible by the batch size. The generator is expected to loop over its data indefinitely. An epoch finishes when `steps_per_epoch` batches have been seen by the model.
- **steps_per_epoch**: Integer. Total number of steps (batches of samples) to yield from `generator` before declaring one epoch finished and starting the next epoch. It should typically be equal to the number of samples of your dataset divided by the batch size. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire data provided, as defined by `steps_per_epoch`. Note that in conjunction with `initial_epoch`, `epochs` is to be understood

as “final epoch”. The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.

- **verbose**: Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **validation_data**: This can be either
- a generator for the validation data
- tuple `(x_val, y_val)`
- tuple `(x_val, y_val, val_sample_weights)` on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data.
- **validation_steps**: Only relevant if `validation_data` is a generator. Total number of steps (batches of samples) to yield from `validation_data` generator before stopping at the end of every epoch. It should typically be equal to the number of samples of your validation dataset divided by the batch size. Optional for `Sequence`: if unspecified, will use the `len(validation_data)` as a number of steps.
- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to “pay more attention” to samples from an under-represented class.
- **max_queue_size**: Integer. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers**: Integer. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: Boolean. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can’t be passed easily to children processes.
- **shuffle**: Boolean. Whether to shuffle the order of the batches at the beginning of each epoch. Only used with instances of `Sequence` (`keras.utils.Sequence`). Has no effect when `steps_per_epoch` is not `None`.
- **initial_epoch**: Integer. Epoch at which to start training (useful for resuming a previous training run).

Returns

A `History` object. Its `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

Raises

- **ValueError**: In case the generator yields data in an invalid format.

Example

```
def generate_arrays_from_file(path):
    while True:
        with open(path) as f:
            for line in f:
                # create numpy arrays of input data
                # and labels, from each line in the file
                x1, x2, y = process_line(line)
                yield ({'input_1': x1, 'input_2': x2}, {'output': y})

model.fit_generator(generate_arrays_from_file('/my_file.txt'),
                    steps_per_epoch=10000, epochs=10)
```

---

evaluate_generator

```
evaluate_generator(self, generator, steps=None, max_queue_size=10, workers=1, use_multiproce
```

Evaluates the model on a data generator.

The generator should return the same kind of data as accepted by `test_on_batch`.

Arguments

- **generator**: Generator yielding tuples (inputs, targets) or (inputs, targets, sample_weights) or an instance of Sequence (keras.utils.Sequence) object in order to avoid duplicate data when using multiprocessing.
- **steps**: Total number of steps (batches of samples) to yield from `generator` before stopping. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **max_queue_size**: maximum size for the generator queue
- **workers**: Integer. Maximum number of processes to spin up when using process based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: if True, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non picklable arguments to the generator as they can't be passed easily to children processes.
- **verbose**: verbosity mode, 0 or 1.

Returns

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

Raises

- **ValueError**: In case the generator yields data in an invalid format.

---

predict_generator

```
predict_generator(self, generator, steps=None, max_queue_size=10, workers=1, use_multiproces
```

Generates predictions for the input samples from a data generator.

The generator should return the same kind of data as accepted by `predict_on_batch`.

Arguments

- **generator**: Generator yielding batches of input samples or an instance of Sequence (keras.utils.Sequence) object in order to avoid duplicate data when using multiprocessing.
- **steps**: Total number of steps (batches of samples) to yield from `generator` before stopping. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **max_queue_size**: Maximum size for the generator queue.
- **workers**: Integer. Maximum number of processes to spin up when using process based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: If `True`, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non picklable arguments to the generator as they can't be passed easily to children processes.
- **verbose**: verbosity mode, 0 or 1.

Returns

Numpy array(s) of predictions.

Raises

- **ValueError**: In case the generator yields data in an invalid format.

---

get_layer

```
get_layer(self, name=None, index=None)
```

Retrieves a layer based on either its name (unique) or index.

If `name` and `index` are both provided, `index` will take precedence.

Indices are based on order of horizontal graph traversal (bottom-up).

Arguments

- **name**: String, name of layer.
- **index**: Integer, index of layer.

Returns

A layer instance.

Raises

- **ValueError**: In case of invalid layer name or index.

Model (functional API)

Model class API

In the functional API, given some input tensor(s) and output tensor(s), you can instantiate a `Model` via:

```
from keras.models import Model
from keras.layers import Input, Dense

a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(inputs=a, outputs=b)
```

This model will include all layers required in the computation of `b` given `a`.

In the case of multi-input or multi-output models, you can use lists as well:

```
model = Model(inputs=[a1, a2], outputs=[b1, b2, b3])
```

For a detailed introduction of what `Model` can do, read this guide to the Keras functional API.

Useful attributes of Model

- `model.layers` is a flattened list of the layers comprising the model graph.
- `model.inputs` is the list of input tensors.
- `model.outputs` is the list of output tensors.

Methods

compile

```
compile(self, optimizer, loss=None, metrics=None, loss_weights=None, sample_weight_mode=None
```

Configures the model for training.

Arguments

- **optimizer**: String (name of optimizer) or optimizer instance. See optimizers.
- **loss**: String (name of objective function) or objective function. See losses. If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of losses. The loss value that will be minimized by the model will then be the sum of all individual losses.
- **metrics**: List of metrics to be evaluated by the model during training and testing. Typically you will use `metrics=['accuracy']`. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as `metrics={'output_a': 'accuracy'}`.
- **loss_weights**: Optional list or dictionary specifying scalar coefficients (Python floats) to weight the loss contributions of different model outputs. The loss value that will be minimized by the model will then be the *weighted sum* of all individual losses, weighted by the `loss_weights` coefficients. If a list, it is expected to have a 1:1 mapping to the model's outputs. If a tensor, it is expected to map output names (strings) to scalar coefficients.
- **sample_weight_mode**: If you need to do timestep-wise sample weighting (2D weights), set this to `"temporal"`. `None` defaults to sample-wise weights (1D). If the model has multiple outputs, you can use a different `sample_weight_mode` on each output by passing a dictionary or a list of modes.
- **weighted_metrics**: List of metrics to be evaluated and weighted by sample_weight or class_weight during training and testing.
- **target_tensors**: By default, Keras will create placeholders for the model's target, which will be fed with the target data during training. If instead you would like to use your own target tensors (in turn, Keras will not expect external Numpy data for these targets at training time), you can specify them via the `target_tensors` argument. It can be a single tensor (for a single-output model), a list of tensors, or a dict mapping output names to target tensors.
- ___**kwargs___: When using the Theano/CNTK backends, these arguments are passed into `K.function`. When using the TensorFlow backend, these arguments are passed into `tf.Session.run`.

Raises

- **ValueError**: In case of invalid arguments for `optimizer`, `loss`, `metrics` or `sample_weight_mode`.

---

fit

```
fit(self, x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_s
```

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

Arguments

- **x**: Numpy array of training data (if the model has a single input), or list of Numpy arrays (if the model has multiple inputs). If input layers in the model are named, you can also pass a dictionary mapping input names to Numpy arrays. `x` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **y**: Numpy array of target (label) data (if the model has a single output), or list of Numpy arrays (if the model has multiple outputs). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays. `y` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **batch_size**: Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32.
- **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided. Note that in conjunction with `initial_epoch`, `epochs` is to be understood as "final epoch". The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.
- **verbose**: Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **validation_split**: Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data is selected from the last samples in the `x` and `y` data provided, before shuffling.
- **validation_data**: tuple `(x_val, y_val)` or tuple `(x_val, y_val, val_sample_weights)` on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. `validation_data` will override `validation_split`.
- **shuffle**: Boolean (whether to shuffle the training data before each epoch) or str (for 'batch'). 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when `steps_per_epoch` is not `None`.
- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class.
- **sample_weight**: Optional Numpy array of weights for the training samples, used for weighting the loss function (during training only). You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape `(samples, sequence_length)`,

to apply a different weight to every timestep of every sample. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.
- **initial_epoch**: Integer. Epoch at which to start training (useful for resuming a previous training run).
- **steps_per_epoch**: Integer or `None`. Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. When training with input tensors such as TensorFlow data tensors, the default `None` is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined.
- **validation_steps**: Only relevant if `steps_per_epoch` is specified. Total number of steps (batches of samples) to validate before stopping.

Returns

A `History` object. Its `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

Raises

- **RuntimeError**: If the model was never compiled.
- **ValueError**: In case of mismatch between the provided input data and what the model expects.

---

evaluate

```
evaluate(self, x=None, y=None, batch_size=None, verbose=1, sample_weight=None, steps=None)
```

Returns the loss value & metrics values for the model in test mode.

Computation is done in batches.

Arguments

- **x**: Numpy array of test data (if the model has a single input), or list of Numpy arrays (if the model has multiple inputs). If input layers in the model are named, you can also pass a dictionary mapping input names to Numpy arrays. `x` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **y**: Numpy array of target (label) data (if the model has a single output), or list of Numpy arrays (if the model has multiple outputs). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays. `y` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **batch_size**: Integer or `None`. Number of samples per evaluation step. If unspecified, `batch_size` will default to 32.
- **verbose**: 0 or 1. Verbosity mode. 0 = silent, 1 = progress bar.

- **sample_weight**: Optional Numpy array of weights for the test samples, used for weighting the loss function. You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape `(samples, sequence_length)`, to apply a different weight to every timestep of every sample. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.
- **steps**: Integer or `None`. Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of `None`.

Returns

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

predict

```
predict(self, x, batch_size=None, verbose=0, steps=None)
```

Generates output predictions for the input samples.

Computation is done in batches.

Arguments

- **x**: The input data, as a Numpy array (or list of Numpy arrays if the model has multiple outputs).
- **batch_size**: Integer. If unspecified, it will default to 32.
- **verbose**: Verbosity mode, 0 or 1.
- **steps**: Total number of steps (batches of samples) before declaring the prediction round finished. Ignored with the default value of `None`.

Returns

Numpy array(s) of predictions.

Raises

- **ValueError**: In case of mismatch between the provided input data and the model's expectations, or in case a stateful model receives a number of samples that is not a multiple of the batch size.

---

train_on_batch

```
train_on_batch(self, x, y, sample_weight=None, class_weight=None)
```

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

Arguments

- **x**: Numpy array of training data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **sample_weight**: Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify sample_weight_mode="temporal" in compile().
- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) to apply to the model's loss for the samples from this class during training. This can be useful to tell the model to "pay more attention" to samples from an under-represented class.

Returns

Scalar training loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

test_on_batch

```
test_on_batch(self, x, y, sample_weight=None)
```

Test the model on a single batch of samples.

Arguments

- **x**: Numpy array of test data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **sample_weight**: Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify sample_weight_mode="temporal" in compile().

Returns

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

predict_on_batch

```
predict_on_batch(self, x)
```

Returns predictions for a single batch of samples.

Arguments

- **x**: Input samples, as a Numpy array.

Returns

Numpy array(s) of predictions.

---

fit_generator

```
fit_generator(self, generator, steps_per_epoch=None, epochs=1, verbose=1, callbacks=None, va
```

Trains the model on data generated batch-by-batch by a Python generator (or an instance of `Sequence`).

The generator is run in parallel to the model, for efficiency. For instance, this allows you to do real-time data augmentation on images on CPU in parallel to training your model on GPU.

The use of `keras.utils.Sequence` guarantees the ordering and guarantees the single use of every input per epoch when using `use_multiprocessing=True`.

Arguments

- **generator**: A generator or an instance of `Sequence` (`keras.utils.Sequence`) object in order to avoid duplicate data when using multiprocessing. The output of the generator must be either
- a tuple `(inputs, targets)`
- a tuple `(inputs, targets, sample_weights)`. This tuple (a single output of the generator) makes a single batch. Therefore, all arrays in this tuple must have the same length (equal to the size of this batch). Different batches may have different sizes. For example, the last batch of the epoch is commonly smaller than the others, if the size of the dataset is not divisible by the batch size. The generator is expected to loop over its data indefinitely. An epoch finishes when `steps_per_epoch` batches have been seen by the model.

- **steps_per_epoch**: Integer. Total number of steps (batches of samples) to yield from `generator` before declaring one epoch finished and starting the next epoch. It should typically be equal to the number of samples of your dataset divided by the batch size. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire data provided, as defined by `steps_per_epoch`. Note that in conjunction with `initial_epoch`, `epochs` is to be understood as "final epoch". The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.
- **verbose**: Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **validation_data**: This can be either
- a generator for the validation data
- tuple `(x_val, y_val)`
- tuple `(x_val, y_val, val_sample_weights)` on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data.
- **validation_steps**: Only relevant if `validation_data` is a generator. Total number of steps (batches of samples) to yield from `validation_data` generator before stopping at the end of every epoch. It should typically be equal to the number of samples of your validation dataset divided by the batch size. Optional for `Sequence`: if unspecified, will use the `len(validation_data)` as a number of steps.
- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class.
- **max_queue_size**: Integer. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers**: Integer. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: Boolean. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.
- **shuffle**: Boolean. Whether to shuffle the order of the batches at the beginning of each epoch. Only used with instances of `Sequence` (`keras.utils.Sequence`). Has no effect when `steps_per_epoch` is not `None`.
- **initial_epoch**: Integer. Epoch at which to start training (useful for resuming a previous training run).

Returns

A `History` object. Its `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

Raises

- **ValueError**: In case the generator yields data in an invalid format.

Example

```
def generate_arrays_from_file(path):
    while True:
        with open(path) as f:
            for line in f:
                # create numpy arrays of input data
                # and labels, from each line in the file
                x1, x2, y = process_line(line)
                yield ({'input_1': x1, 'input_2': x2}, {'output': y})

model.fit_generator(generate_arrays_from_file('/my_file.txt'),
                    steps_per_epoch=10000, epochs=10)
```

---

evaluate_generator

```
evaluate_generator(self, generator, steps=None, max_queue_size=10, workers=1, use_multiproce
```

Evaluates the model on a data generator.

The generator should return the same kind of data as accepted by `test_on_batch`.

Arguments

- **generator**: Generator yielding tuples (inputs, targets) or (inputs, targets, sample_weights) or an instance of Sequence (keras.utils.Sequence) object in order to avoid duplicate data when using multiprocessing.
- **steps**: Total number of steps (batches of samples) to yield from `generator` before stopping. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **max_queue_size**: maximum size for the generator queue
- **workers**: Integer. Maximum number of processes to spin up when using process based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: if True, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass

non picklable arguments to the generator as they can't be passed easily to children processes.
- **verbose**: verbosity mode, 0 or 1.

Returns

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

Raises

- **ValueError**: In case the generator yields data in an invalid format.

---

predict_generator

```
predict_generator(self, generator, steps=None, max_queue_size=10, workers=1, use_multiproces
```

Generates predictions for the input samples from a data generator.

The generator should return the same kind of data as accepted by `predict_on_batch`.

Arguments

- **generator**: Generator yielding batches of input samples or an instance of Sequence (keras.utils.Sequence) object in order to avoid duplicate data when using multiprocessing.
- **steps**: Total number of steps (batches of samples) to yield from `generator` before stopping. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **max_queue_size**: Maximum size for the generator queue.
- **workers**: Integer. Maximum number of processes to spin up when using process based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: If `True`, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non picklable arguments to the generator as they can't be passed easily to children processes.
- **verbose**: verbosity mode, 0 or 1.

Returns

Numpy array(s) of predictions.

Raises

- **ValueError**: In case the generator yields data in an invalid format.

---

get_layer

```
get_layer(self, name=None, index=None)
```

Retrieves a layer based on either its name (unique) or index.

If `name` and `index` are both provided, `index` will take precedence.

Indices are based on order of horizontal graph traversal (bottom-up).

Arguments

- **name**: String, name of layer.
- **index**: Integer, index of layer.

Returns

A layer instance.

Raises

- **ValueError**: In case of invalid layer name or index.

About Keras layers

About Keras layers

All Keras layers have a number of methods in common:

- `layer.get_weights()`: returns the weights of the layer as a list of Numpy arrays.
- `layer.set_weights(weights)`: sets the weights of the layer from a list of Numpy arrays (with the same shapes as the output of `get_weights`).
- `layer.get_config()`: returns a dictionary containing the configuration of the layer. The layer can be reinstantiated from its config via:

```
layer = Dense(32)
config = layer.get_config()
reconstructed_layer = Dense.from_config(config)
```

Or:

```
from keras import layers

config = layer.get_config()
layer = layers.deserialize({'class_name': layer.__class__.__name__,
                            'config': config})
```

If a layer has a single node (i.e. if it isn't a shared layer), you can get its input tensor, output tensor, input shape and output shape via:

- `layer.input`
- `layer.output`

- `layer.input_shape`
- `layer.output_shape`

If the layer has multiple nodes (see: the concept of layer node and shared layers), you can use the following methods:

- `layer.get_input_at(node_index)`
- `layer.get_output_at(node_index)`
- `layer.get_input_shape_at(node_index)`
- `layer.get_output_shape_at(node_index)`

Core Layers

[source] ##### Dense

```
keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniforn
```

Just your regular densely-connected NN layer.

`Dense` implements the operation: `output = activation(dot(input, kernel) + bias)` where `activation` is the element-wise activation function passed as the `activation` argument, `kernel` is a weights matrix created by the layer, and `bias` is a bias vector created by the layer (only applicable if `use_bias` is `True`).

- **Note**: if the input to the layer has a rank greater than 2, then it is flattened prior to the initial dot product with `kernel`.

Example

```
### as first layer in a sequential model:
model = Sequential()
model.add(Dense(32, input_shape=(16,)))
### now the model will take as input arrays of shape (*, 16)
### and output arrays of shape (*, 32)

### after the first layer, you don't need to specify
### the size of the input anymore:
model.add(Dense(32))
```

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).

- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

nD tensor with shape: `(batch_size, ..., input_dim)`. The most common situation would be a 2D input with shape `(batch_size, input_dim)`.

Output shape

nD tensor with shape: `(batch_size, ..., units)`. For instance, for a 2D input with shape `(batch_size, input_dim)`, the output would have shape `(batch_size, units)`.

[source] ##### Activation

```
keras.layers.Activation(activation)
```

Applies an activation function to an output.

Arguments

- **activation**: name of activation function to use (see: activations), or alternatively, a Theano or TensorFlow operation.

Input shape

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

Output shape

Same shape as input.

[source] ##### Dropout

```
keras.layers.Dropout(rate, noise_shape=None, seed=None)
```

Applies Dropout to the input.

Dropout consists in randomly setting a fraction `rate` of input units to 0 at each update during training time, which helps prevent overfitting.

Arguments

- **rate**: float between 0 and 1. Fraction of the input units to drop.
- **noise_shape**: 1D integer tensor representing the shape of the binary dropout mask that will be multiplied with the input. For instance, if your inputs have shape `(batch_size, timesteps, features)` and you want the dropout mask to be the same for all timesteps, you can use `noise_shape=(batch_size, 1, features)`.
- **seed**: A Python integer to use as random seed.

References

- Dropout: A Simple Way to Prevent Neural Networks from Overfitting

---

[source] ##### Flatten

```
keras.layers.Flatten(data_format='channels_last')
```

Flattens the input. Does not affect the batch size.

Arguments

- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, ..., channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, ...)`.

Example

```
model = Sequential()
model.add(Conv2D(64, 3, 3,
                 border_mode='same',
                 input_shape=(3, 32, 32)))
### now: model.output_shape == (None, 64, 32, 32)

model.add(Flatten())
### now: model.output_shape == (None, 65536)
```

[source] ##### Input

```
keras.engine.input_layer.Input()
```

`Input()` is used to instantiate a Keras tensor.

A Keras tensor is a tensor object from the underlying backend (Theano, TensorFlow or CNTK), which we augment with certain attributes that allow us to build a Keras model just by knowing the inputs and outputs of the model.

For instance, if a, b and c are Keras tensors, it becomes possible to do: `model = Model(input=[a, b], output=c)`

The added Keras attributes are: - `_keras_shape`: Integer shape tuple propagated via Keras-side shape inference. - `_keras_history`: Last layer applied to the tensor. the entire layer graph is retrievable from that layer, recursively.

Arguments

- **shape**: A shape tuple (integer), not including the batch size. For instance, `shape=(32,)` indicates that the expected input will be batches of 32-dimensional vectors.
- **batch_shape**: A shape tuple (integer), including the batch size. For instance, `batch_shape=(10, 32)` indicates that the expected input will be batches of 10 32-dimensional vectors. `batch_shape=(None, 32)` indicates batches of an arbitrary number of 32-dimensional vectors.
- **name**: An optional name string for the layer. Should be unique in a model (do not reuse the same name twice). It will be autogenerated if it isn't provided.
- **dtype**: The data type expected by the input, as a string (`float32`, `float64`, `int32`…)
- **sparse**: A boolean specifying whether the placeholder to be created is sparse.
- **tensor**: Optional existing tensor to wrap into the `Input` layer. If set, the layer will not create a placeholder tensor.

Returns

A tensor.

Example

```
### this is a logistic regression in Keras
x = Input(shape=(32,))
y = Dense(16, activation='softmax')(x)
model = Model(x, y)
```

[source] ##### Reshape

`keras.layers.Reshape(target_shape)`

Reshapes an output to a certain shape.

Arguments

- **target_shape**: target shape. Tuple of integers. Does not include the batch axis.

Input shape

Arbitrary, although all dimensions in the input shaped must be fixed. Use the keyword argument `input_shape` (tuple of integers, does not include the batch axis) when using this layer as the first layer in a model.

Output shape

`(batch_size,) + target_shape`

Example

```
### as first layer in a Sequential model
model = Sequential()
model.add(Reshape((3, 4), input_shape=(12,)))
### now: model.output_shape == (None, 3, 4)
### note: `None` is the batch dimension

### as intermediate layer in a Sequential model
model.add(Reshape((6, 2)))
### now: model.output_shape == (None, 6, 2)

### also supports shape inference using `-1` as dimension
model.add(Reshape((-1, 2, 2)))
### now: model.output_shape == (None, 3, 2, 2)
```

[source] ##### Permute

`keras.layers.Permute(dims)`

Permutes the dimensions of the input according to a given pattern.

Useful for e.g. connecting RNNs and convnets together.

Example

```
model = Sequential()
model.add(Permute((2, 1), input_shape=(10, 64)))
### now: model.output_shape == (None, 64, 10)
### note: `None` is the batch dimension
```

Arguments

- **dims**: Tuple of integers. Permutation pattern, does not include the samples dimension. Indexing starts at 1. For instance, `(2, 1)` permutes the first and second dimension of the input.

Input shape

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

Output shape

Same as the input shape, but with the dimensions re-ordered according to the specified pattern.

[source] ##### RepeatVector

`keras.layers.RepeatVector(n)`

Repeats the input n times.

Example

```
model = Sequential()
model.add(Dense(32, input_dim=32))
### now: model.output_shape == (None, 32)
### note: `None` is the batch dimension

model.add(RepeatVector(3))
### now: model.output_shape == (None, 3, 32)
```

Arguments

- **n**: integer, repetition factor.

Input shape

2D tensor of shape `(num_samples, features)`.

Output shape

3D tensor of shape `(num_samples, n, features)`.

[source] ##### Lambda

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

Wraps arbitrary expression as a `Layer` object.

Examples

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

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

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

def antirectifier_output_shape(input_shape):
    shape = list(input_shape)
    assert len(shape) == 2  # only valid for 2D tensors
    shape[-1] *= 2
    return tuple(shape)

model.add(Lambda(antirectifier,
                 output_shape=antirectifier_output_shape))
```

Arguments

- **function**: The function to be evaluated. Takes input tensor as first argument.
- **output_shape**: Expected output shape from function. Only relevant when using Theano. Can be a tuple or function. If a tuple, it only specifies the first dimension onward; sample dimension is assumed either the same as the input: `output_shape = (input_shape[0], ) + output_shape` or, the input is `None` and the sample dimension is also `None`: `output_shape = (None, ) + output_shape` If a function, it specifies the entire shape as a function of the input shape: `output_shape = f(input_shape)`
- **arguments**: optional dictionary of keyword arguments to be passed to the function.

Input shape

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

Output shape

Specified by `output_shape` argument (or auto-inferred when using TensorFlow).

[source] ##### ActivityRegularization

```
keras.layers.ActivityRegularization(l1=0.0, l2=0.0)
```

Layer that applies an update to the cost function based input activity.

Arguments

- **l1**: L1 regularization factor (positive float).
- **l2**: L2 regularization factor (positive float).

Input shape

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

Output shape

Same shape as input.

[source] ##### Masking

```
keras.layers.Masking(mask_value=0.0)
```

Masks a sequence by using a mask value to skip timesteps.

For each timestep in the input tensor (dimension #1 in the tensor), if all values in the input tensor at that timestep are equal to `mask_value`, then the timestep will be masked (skipped) in all downstream layers (as long as they support masking).

If any downstream layer does not support masking yet receives such an input mask, an exception will be raised.

Example

Consider a Numpy data array `x` of shape `(samples, timesteps, features)`, to be fed to an LSTM layer. You want to mask timestep #3 and #5 because you lack data for these timesteps. You can:

- set `x[:, 3, :] = 0.` and `x[:, 5, :] = 0.`
- insert a `Masking` layer with `mask_value=0.` before the LSTM layer:

```
model = Sequential()
model.add(Masking(mask_value=0., input_shape=(timesteps, features)))
model.add(LSTM(32))
```

Convolutional Layers

[source] ##### Conv1D

```
keras.layers.Conv1D(filters, kernel_size, strides=1, padding='valid', dilation_rate=1, acti
```

1D convolution layer (e.g. temporal convolution).

This layer creates a convolution kernel that is convolved with the layer input over a single spatial (or temporal) dimension to produce a tensor of outputs. If `use_bias` is True, a bias vector is created and added to the outputs. Finally, if `activation` is not `None`, it is applied to the outputs as well.

When using this layer as the first layer in a model, provide an `input_shape` argument (tuple of integers or `None`, e.g. `(10, 128)` for sequences of 10 vectors of 128-dimensional vectors, or `(None, 128)` for variable-length sequences of 128-dimensional vectors.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of a single integer, specifying the length of the 1D convolution window.

- **strides**: An integer or tuple/list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: One of `"valid"`, `"causal"` or `"same"` (case-insensitive). `"valid"` means "no padding". `"same"` results in padding the input such that the output has the same length as the original input. `"causal"` results in causal (dilated) convolutions, e.g. output[t] does not depend on input[t+1:]. Useful when modeling temporal data where the model should not violate the temporal order. See WaveNet: A Generative Model for Raw Audio, section 2.1.
- **dilation_rate**: an integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any `strides` value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

3D tensor with shape: `(batch_size, steps, input_dim)`

Output shape

3D tensor with shape: `(batch_size, new_steps, filters)` `steps` value might have changed due to padding or strides.

[source] ##### Conv2D

```
keras.layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None,
```

2D convolution layer (e.g. spatial convolution over images).

This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. If `use_bias` is True, a bias vector is created and added to the outputs. Finally, if `activation` is not `None`, it is applied to the outputs as well.

When using this layer as the first layer in a model, provide the keyword argument `input_shape` (tuple of integers, does not include the sample axis), e.g. `input_shape=(128, 128, 3)` for 128x128 RGB pictures in `data_format="channels_last"`.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive). Note that `"same"` is slightly inconsistent across backends with `strides` != 1, as described here
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).

- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

4D tensor with shape: `(samples, channels, rows, cols)` if data_format='channels_first' or 4D tensor with shape: `(samples, rows, cols, channels)` if data_format='channels_last'.

Output shape

4D tensor with shape: `(samples, filters, new_rows, new_cols)` if data_format='channels_first' or 4D tensor with shape: `(samples, new_rows, new_cols, filters)` if data_format='channels_last'. `rows` and `cols` values might have changed due to padding.

[source] ##### SeparableConv2D

```
keras.layers.SeparableConv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_fo
```

Depthwise separable 2D convolution.

Separable convolutions consist in first performing a depthwise spatial convolution (which acts on each input channel separately) followed by a pointwise convolution which mixes together the resulting output channels. The `depth_multiplier` argument controls how many output channels are generated per input channel in the depthwise step.

Intuitively, separable convolutions can be understood as a way to factorize a convolution kernel into two smaller kernels, or as an extreme version of an Inception block.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs.

`channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

- **dilation_rate**: An integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any `strides` value != 1.
- **depth_multiplier**: The number of depthwise convolution output channels for each input channel. The total number of depthwise convolution output channels will be equal to `filters_in * depth_multiplier`.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **depthwise_initializer**: Initializer for the depthwise kernel matrix (see initializers).
- **pointwise_initializer**: Initializer for the pointwise kernel matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **depthwise_regularizer**: Regularizer function applied to the depthwise kernel matrix (see regularizer).
- **pointwise_regularizer**: Regularizer function applied to the pointwise kernel matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **depthwise_constraint**: Constraint function applied to the depthwise kernel matrix (see constraints).
- **pointwise_constraint**: Constraint function applied to the pointwise kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

4D tensor with shape: `(batch, channels, rows, cols)` if data_format='channels_first' or 4D tensor with shape: `(batch, rows, cols, channels)` if data_format='channels_last'.

Output shape

4D tensor with shape: `(batch, filters, new_rows, new_cols)` if data_format='channels_first' or 4D tensor with shape: `(batch, new_rows, new_cols, filters)` if data_format='channels_last'. `rows` and `cols` values might have changed due to padding.

[source] ##### Conv2DTranspose

```
keras.layers.Conv2DTranspose(filters, kernel_size, strides=(1, 1), padding='valid', data_fo
```

Transposed convolution layer (sometimes called Deconvolution).

The need for transposed convolutions generally arises from the desire to use a transformation going in the opposite direction of a normal convolution, i.e., from something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution.

When using this layer as the first layer in a model, provide the keyword argument `input_shape` (tuple of integers, does not include the sample axis), e.g. `input_shape=(128, 128, 3)` for 128x128 RGB pictures in `data_format="channels_last"`.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.

- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

4D tensor with shape: `(batch, channels, rows, cols)` if data_format='channels_first' or 4D tensor with shape: `(batch, rows, cols, channels)` if data_format='channels_last'.

Output shape

4D tensor with shape: `(batch, filters, new_rows, new_cols)` if data_format='channels_first' or 4D tensor with shape: `(batch, new_rows, new_cols, filters)` if data_format='channels_last'. `rows` and `cols` values might have changed due to padding.

References

- A guide to convolution arithmetic for deep learning
- Deconvolutional Networks

[source] ##### Conv3D

```
keras.layers.Conv3D(filters, kernel_size, strides=(1, 1, 1), padding='valid', data_format=No
```

3D convolution layer (e.g. spatial convolution over volumes).

This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. If `use_bias` is True, a bias vector is created and added to the outputs. Finally, if `activation` is not `None`, it is applied to the outputs as well.

When using this layer as the first layer in a model, provide the keyword argument `input_shape` (tuple of integers, does not include the sample axis), e.g. `input_shape=(128, 128, 128, 1)` for 128x128x128 volumes with a single channel, in `data_format="channels_last"`.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).

- **kernel_size**: An integer or tuple/list of 3 integers, specifying the depth, height and width of the 3D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 3 integers, specifying the strides of the convolution along each spatial dimension. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: an integer or tuple/list of 3 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

5D tensor with shape: `(samples, channels, conv_dim1, conv_dim2, conv_dim3)` if data_format='channels_first' or 5D tensor with shape: `(samples, conv_dim1, conv_dim2, conv_dim3, channels)` if data_format='channels_last'.

Output shape

5D tensor with shape: `(samples, filters, new_conv_dim1, new_conv_dim2,` `new_conv_dim3)` if data_format='channels_first' or 5D tensor with shape: `(samples, new_conv_dim1, new_conv_dim2, new_conv_dim3, filters)` if data_format='channels_last'. `new_conv_dim1`, `new_conv_dim2` and `new_conv_dim3` values might have changed due to padding.

[source] ##### Cropping1D

```
keras.layers.Cropping1D(cropping=(1, 1))
```

Cropping layer for 1D input (e.g. temporal sequence).

It crops along the time dimension (axis 1).

Arguments

- **cropping**: int or tuple of int (length 2) How many units should be trimmed off at the beginning and end of the cropping dimension (axis 1). If a single int is provided, the same value will be used for both.

Input shape

3D tensor with shape `(batch, axis_to_crop, features)`

Output shape

3D tensor with shape `(batch, cropped_axis, features)`

[source] ##### Cropping2D

```
keras.layers.Cropping2D(cropping=((0, 0), (0, 0)), data_format=None)
```

Cropping layer for 2D input (e.g. picture).

It crops along spatial dimensions, i.e. width and height.

Arguments

- **cropping**: int, or tuple of 2 ints, or tuple of 2 tuples of 2 ints.
- If int: the same symmetric cropping is applied to width and height.
- If tuple of 2 ints: interpreted as two different symmetric cropping values for height and width: `(symmetric_height_crop, symmetric_width_crop)`.
- If tuple of 2 tuples of 2 ints: interpreted as `((top_crop, bottom_crop), (left_crop, right_crop))`
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

4D tensor with shape: - If `data_format` is `"channels_last"`: `(batch, rows, cols, channels)` - If `data_format` is `"channels_first"`: `(batch, channels, rows, cols)`

Output shape

4D tensor with shape: - If `data_format` is `"channels_last"`: `(batch, cropped_rows, cropped_cols, channels)` - If `data_format` is `"channels_first"`: `(batch, channels, cropped_rows, cropped_cols)`

Examples

```
### Crop the input 2D images or feature maps
model = Sequential()
model.add(Cropping2D(cropping=((2, 2), (4, 4)),
                     input_shape=(28, 28, 3)))
### now model.output_shape == (None, 24, 20, 3)
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Cropping2D(cropping=((2, 2), (2, 2))))
### now model.output_shape == (None, 20, 16. 64)
```

[source] ##### Cropping3D

`keras.layers.Cropping3D(cropping=((1, 1), (1, 1), (1, 1)), data_format=None)`

Cropping layer for 3D data (e.g. spatial or spatio-temporal).

Arguments

- **cropping**: int, or tuple of 3 ints, or tuple of 3 tuples of 2 ints.
- If int: the same symmetric cropping is applied to depth, height, and width.
- If tuple of 3 ints: interpreted as two different symmetric cropping values for depth, height, and width: `(symmetric_dim1_crop, symmetric_dim2_crop, symmetric_dim3_crop)`.
- If tuple of 3 tuples of 2 ints: interpreted as `((left_dim1_crop, right_dim1_crop), (left_dim2_crop, right_dim2_crop), (left_dim3_crop, right_dim3_crop))`
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

5D tensor with shape: - If `data_format` is `"channels_last"`: `(batch, first_axis_to_crop, second_axis_to_crop, third_axis_to_crop, depth)` - If `data_format` is `"channels_first"`: `(batch, depth, first_axis_to_crop, second_axis_to_crop, third_axis_to_crop)`

Output shape

5D tensor with shape: - If `data_format` is `"channels_last"`: `(batch, first_cropped_axis, second_cropped_axis, third_cropped_axis, depth)` - If `data_format` is `"channels_first"`: `(batch, depth, first_cropped_axis, second_cropped_axis, third_cropped_axis)`

[source] ##### UpSampling1D

```
keras.layers.UpSampling1D(size=2)
```

Upsampling layer for 1D inputs.

Repeats each temporal step `size` times along the time axis.

Arguments

- **size**: integer. Upsampling factor.

Input shape

3D tensor with shape: `(batch, steps, features)`.

Output shape

3D tensor with shape: `(batch, upsampled_steps, features)`.

[source] ##### UpSampling2D

```
keras.layers.UpSampling2D(size=(2, 2), data_format=None)
```

Upsampling layer for 2D inputs.

Repeats the rows and columns of the data by size[0] and size[1] respectively.

Arguments

- **size**: int, or tuple of 2 integers. The upsampling factors for rows and columns.
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

4D tensor with shape: - If `data_format` is `"channels_last"`: `(batch, rows, cols, channels)` - If `data_format` is `"channels_first"`: `(batch, channels, rows, cols)`

Output shape

4D tensor with shape: - If `data_format` is `"channels_last"`: `(batch, upsampled_rows, upsampled_cols, channels)` - If `data_format` is `"channels_first"`: `(batch, channels, upsampled_rows, upsampled_cols)`

[source] ##### UpSampling3D

```
keras.layers.UpSampling3D(size=(2, 2, 2), data_format=None)
```

Upsampling layer for 3D inputs.

Repeats the 1st, 2nd and 3rd dimensions of the data by size[0], size[1] and size[2] respectively.

Arguments

- **size**: int, or tuple of 3 integers. The upsampling factors for dim1, dim2 and dim3.
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

5D tensor with shape: - If `data_format` is `"channels_last"`: `(batch, dim1, dim2, dim3, channels)` - If `data_format` is `"channels_first"`: `(batch, channels, dim1, dim2, dim3)`

Output shape

5D tensor with shape: - If `data_format` is `"channels_last"`: `(batch, upsampled_dim1, upsampled_dim2, upsampled_dim3, channels)` - If `data_format` is `"channels_first"`: `(batch, channels, upsampled_dim1, upsampled_dim2, upsampled_dim3)`

[source] ##### ZeroPadding1D

```
keras.layers.ZeroPadding1D(padding=1)
```

Zero-padding layer for 1D input (e.g. temporal sequence).

Arguments

- **padding**: int, or tuple of int (length 2), or dictionary.
- If int: How many zeros to add at the beginning and end of the padding dimension (axis 1).
- If tuple of int (length 2): How many zeros to add at the beginning and at the end of the padding dimension (`(left_pad, right_pad)`).

Input shape

3D tensor with shape `(batch, axis_to_pad, features)`

Output shape

3D tensor with shape `(batch, padded_axis, features)`

[source] ##### ZeroPadding2D

```
keras.layers.ZeroPadding2D(padding=(1, 1), data_format=None)
```

Zero-padding layer for 2D input (e.g. picture).

This layer can add rows and columns of zeros at the top, bottom, left and right side of an image tensor.

Arguments

- **padding**: int, or tuple of 2 ints, or tuple of 2 tuples of 2 ints.
- If int: the same symmetric padding is applied to width and height.
- If tuple of 2 ints: interpreted as two different symmetric padding values for height and width: `(symmetric_height_pad, symmetric_width_pad)`.
- If tuple of 2 tuples of 2 ints: interpreted as `((top_pad, bottom_pad), (left_pad, right_pad))`
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

4D tensor with shape: - If `data_format` is `"channels_last"`: `(batch, rows, cols, channels)` - If `data_format` is `"channels_first"`: `(batch, channels, rows, cols)`

Output shape

4D tensor with shape: - If `data_format` is `"channels_last"`: `(batch, padded_rows, padded_cols, channels)` - If `data_format` is `"channels_first"`: `(batch, channels, padded_rows, padded_cols)`

[source] ##### ZeroPadding3D

```
keras.layers.ZeroPadding3D(padding=(1, 1, 1), data_format=None)
```

Zero-padding layer for 3D data (spatial or spatio-temporal).

Arguments

- **padding**: int, or tuple of 3 ints, or tuple of 3 tuples of 2 ints.
- If int: the same symmetric padding is applied to width and height.
- If tuple of 3 ints: interpreted as two different symmetric padding values for height and width: `(symmetric_dim1_pad, symmetric_dim2_pad, symmetric_dim3_pad)`.
- If tuple of 3 tuples of 2 ints: interpreted as `((left_dim1_pad, right_dim1_pad), (left_dim2_pad, right_dim2_pad), (left_dim3_pad, right_dim3_pad))`
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

5D tensor with shape: - If `data_format` is `"channels_last"`: `(batch, first_axis_to_pad, second_axis_to_pad, third_axis_to_pad, depth)` - If `data_format` is `"channels_first"`: `(batch, depth, first_axis_to_pad, second_axis_to_pad, third_axis_to_pad)`

Output shape

5D tensor with shape: - If `data_format` is `"channels_last"`: `(batch, first_padded_axis, second_padded_axis, third_axis_to_pad, depth)` - If `data_format` is `"channels_first"`: `(batch, depth, first_padded_axis, second_padded_axis, third_axis_to_pad)`

Pooling Layers

[source] ##### MaxPooling1D

```
keras.layers.MaxPooling1D(pool_size=2, strides=None, padding='valid')
```

Max pooling operation for temporal data.

Arguments

- **pool_size**: Integer, size of the max pooling windows.
- **strides**: Integer, or None. Factor by which to downscale. E.g. 2 will halve the input. If None, it will default to `pool_size`.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).

Input shape

3D tensor with shape: `(batch_size, steps, features)`.

Output shape

3D tensor with shape: `(batch_size, downsampled_steps, features)`.

[source] ##### MaxPooling2D

```
keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None
```

Max pooling operation for spatial data.

Arguments

- **pool_size**: integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.
- **strides**: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to `pool_size`.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, rows, cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, rows, cols)`

Output shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, pooled_rows, pooled_cols, channels)`

- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, pooled_rows, pooled_cols)`

[source] ##### MaxPooling3D

```
keras.layers.MaxPooling3D(pool_size=(2, 2, 2), strides=None, padding='valid', data_format=No
```

Max pooling operation for 3D data (spatial or spatio-temporal).

Arguments

- **pool_size**: tuple of 3 integers, factors by which to downscale (dim1, dim2, dim3). (2, 2, 2) will halve the size of the 3D input in each dimension.
- **strides**: tuple of 3 integers, or None. Strides values.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)`

Output shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, pooled_dim1, pooled_dim2, pooled_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, pooled_dim1, pooled_dim2, pooled_dim3)`

[source] ##### AveragePooling1D

```
keras.layers.AveragePooling1D(pool_size=2, strides=None, padding='valid')
```

Average pooling for temporal data.

Arguments

- **pool_size**: Integer, size of the average pooling windows.
- **strides**: Integer, or None. Factor by which to downscale. E.g. 2 will halve the input. If None, it will default to `pool_size`.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).

Input shape

3D tensor with shape: `(batch_size, steps, features)`.

Output shape

3D tensor with shape: `(batch_size, downsampled_steps, features)`.

[source] ##### AveragePooling2D

```
keras.layers.AveragePooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=N
```

Average pooling operation for spatial data.

Arguments

- **pool_size**: integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.
- **strides**: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to `pool_size`.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, rows, cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, rows, cols)`

Output shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, pooled_rows, pooled_cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, pooled_rows, pooled_cols)`

[source] ##### AveragePooling3D

```
keras.layers.AveragePooling3D(pool_size=(2, 2, 2), strides=None, padding='valid', data_forma
```

Average pooling operation for 3D data (spatial or spatio-temporal).

Arguments

- **pool_size**: tuple of 3 integers, factors by which to downscale (dim1, dim2, dim3). (2, 2, 2) will halve the size of the 3D input in each dimension.
- **strides**: tuple of 3 integers, or None. Strides values.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)`

Output shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, pooled_dim1, pooled_dim2, pooled_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, pooled_dim1, pooled_dim2, pooled_dim3)`

[source] ##### GlobalMaxPooling1D

```
keras.layers.GlobalMaxPooling1D()
```

Global max pooling operation for temporal data.

Input shape

3D tensor with shape: `(batch_size, steps, features)`.

Output shape

2D tensor with shape: `(batch_size, features)`

[source] ##### GlobalAveragePooling1D

```
keras.layers.GlobalAveragePooling1D()
```

Global average pooling operation for temporal data.

Input shape

3D tensor with shape: `(batch_size, steps, features)`.

Output shape

2D tensor with shape: `(batch_size, features)`

[source] ##### GlobalMaxPooling2D

```
keras.layers.GlobalMaxPooling2D(data_format=None)
```

Global max pooling operation for spatial data.

Arguments

- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, rows, cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, rows, cols)`

Output shape

2D tensor with shape: `(batch_size, channels)`

[source] ##### GlobalAveragePooling2D

```
keras.layers.GlobalAveragePooling2D(data_format=None)
```

Global average pooling operation for spatial data.

Arguments

- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at

`~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, rows, cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, rows, cols)`

Output shape

2D tensor with shape: `(batch_size, channels)`

Locally-connected Layers

[source] ##### LocallyConnected1D

```
keras.layers.LocallyConnected1D(filters, kernel_size, strides=1, padding='valid', data_forma
```

Locally-connected layer for 1D inputs.

The `LocallyConnected1D` layer works similarly to the `Conv1D` layer, except that weights are unshared, that is, a different set of filters is applied at each different patch of the input.

Example

```
### apply a unshared weight convolution 1d of length 3 to a sequence with
### 10 timesteps, with 64 output filters
model = Sequential()
model.add(LocallyConnected1D(64, 3, input_shape=(10, 32)))
### now model.output_shape == (None, 8, 64)
### add a new conv1d on top
model.add(LocallyConnected1D(32, 3))
### now model.output_shape == (None, 6, 32)
```

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of a single integer, specifying the length of the 1D convolution window.
- **strides**: An integer or tuple/list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: Currently only supports `"valid"` (case-insensitive). `"same"` may be supported in the future.

- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

3D tensor with shape: `(batch_size, steps, input_dim)`

Output shape

3D tensor with shape: `(batch_size, new_steps, filters)` `steps` value might have changed due to padding or strides.

[source] ##### LocallyConnected2D

```
keras.layers.LocallyConnected2D(filters, kernel_size, strides=(1, 1), padding='valid', data_
```

Locally-connected layer for 2D inputs.

The `LocallyConnected2D` layer works similarly to the `Conv2D` layer, except that weights are unshared, that is, a different set of filters is applied at each different patch of the input.

Examples

```
### apply a 3x3 unshared weights convolution with 64 output filters on a 32x32 image
### with `data_format="channels_last"`:
model = Sequential()
model.add(LocallyConnected2D(64, (3, 3), input_shape=(32, 32, 3)))
### now model.output_shape == (None, 30, 30, 64)
### notice that this layer will consume (30*30)*(3*3*3*64) + (30*30)*64 parameters

### add a 3x3 unshared weights convolution on top, with 32 output filters:
model.add(LocallyConnected2D(32, (3, 3)))
### now model.output_shape == (None, 28, 28, 32)
```

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions.
- **padding**: Currently only support `"valid"` (case-insensitive). `"same"` will be supported in future.
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

4D tensor with shape: `(samples, channels, rows, cols)` if data_format='channels_first' or 4D tensor with shape: `(samples, rows, cols, channels)` if data_format='channels_last'.

Output shape

4D tensor with shape: `(samples, filters, new_rows, new_cols)` if data_format='channels_first' or 4D tensor with shape: `(samples, new_rows,` `new_cols, filters)` if data_format='channels_last'. `rows` and `cols` values might have changed due to padding.

Recurrent Layers

[source] ##### RNN

```
keras.layers.RNN(cell, return_sequences=False, return_state=False, go_backwards=False, state
```

Base class for recurrent layers.

Arguments

- **cell**: A RNN cell instance. A RNN cell is a class that has:
- a `call(input_at_t, states_at_t)` method, returning `(output_at_t, states_at_t_plus_1)`. The call method of the cell can also take the optional argument `constants`, see section "Note on passing external constants" below.
- a `state_size` attribute. This can be a single integer (single state) in which case it is the size of the recurrent state (which should be the same as the size of the cell output). This can also be a list/tuple of integers (one size per state). In this case, the first entry (`state_size[0]`) should be the same as the size of the cell output. It is also possible for `cell` to be a list of RNN cell instances, in which cases the cells get stacked on after the other in the RNN, implementing an efficient stacked RNN.
- **return_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence.
- **return_state**: Boolean. Whether to return the last state in addition to the output.
- **go_backwards**: Boolean (default False). If True, process the input sequence backwards and return the reversed sequence.
- **stateful**: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
- **unroll**: Boolean (default False). If True, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.
- **input_dim**: dimensionality of the input (integer). This argument (or alternatively, the keyword argument `input_shape`) is required when using this layer as the first layer in a model.
- **input_length**: Length of input sequences, to be specified when it is constant. This argument is required if you are going to connect `Flatten` then `Dense` layers upstream (without it, the shape of the dense outputs cannot be computed). Note that if the recurrent layer is not the first layer in your model, you would need to specify the input length at the level of the first layer (e.g. via the `input_shape` argument)

Input shape

3D tensor with shape `(batch_size, timesteps, input_dim)`.

Output shape

- if `return_state`: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each with shape `(batch_size, units)`.
- if `return_sequences`: 3D tensor with shape `(batch_size, timesteps, units)`.
- else, 2D tensor with shape `(batch_size, units)`.

Masking

This layer supports masking for input data with a variable number of timesteps. To introduce masks to your data, use an Embedding layer with the `mask_zero` parameter set to `True`.

Note on using statefulness in RNNs

You can set RNN layers to be 'stateful', which means that the states computed for the samples in one batch will be reused as initial states for the samples in the next batch. This assumes a one-to-one mapping between samples in different successive batches.

To enable statefulness: - specify `stateful=True` in the layer constructor. - specify a fixed batch size for your model, by passing if sequential model: `batch_input_shape=(...)` to the first layer in your model. else for functional model with 1 or more Input layers: `batch_shape=(...)` to all the first layers in your model. This is the expected shape of your inputs *including the batch size*. It should be a tuple of integers, e.g. `(32, 10, 100)`. - specify `shuffle=False` when calling fit().

To reset the states of your model, call `.reset_states()` on either a specific layer, or on your entire model.

Note on specifying the initial state of RNNs

You can specify the initial state of RNN layers symbolically by calling them with the keyword argument `initial_state`. The value of `initial_state` should be a tensor or list of tensors representing the initial state of the RNN layer.

You can specify the initial state of RNN layers numerically by calling `reset_states` with the keyword argument `states`. The value of `states` should be a numpy array or list of numpy arrays representing the initial state of the RNN layer.

Note on passing external constants to RNNs

You can pass "external" constants to the cell using the `constants` keyword argument of `RNN.__call__` (as well as `RNN.call`) method. This requires that the `cell.call` method accepts the same keyword argument `constants`. Such constants can be used to condition the cell transformation on additional static inputs (not changing over time), a.k.a. an attention mechanism.

Examples

```
### First, let's define a RNN Cell, as a layer subclass.

class MinimalRNNCell(keras.layers.Layer):

    def __init__(self, units, **kwargs):
        self.units = units
        self.state_size = units
        super(MinimalRNNCell, self).__init__(**kwargs)

    def build(self, input_shape):
        self.kernel = self.add_weight(shape=(input_shape[-1], self.units),
                                      initializer='uniform',
                                      name='kernel')
        self.recurrent_kernel = self.add_weight(
            shape=(self.units, self.units),
            initializer='uniform',
            name='recurrent_kernel')
        self.built = True

    def call(self, inputs, states):
        prev_output = states[0]
        h = K.dot(inputs, self.kernel)
        output = h + K.dot(prev_output, self.recurrent_kernel)
        return output, [output]

### Let's use this cell in a RNN layer:

cell = MinimalRNNCell(32)
x = keras.Input((None, 5))
layer = RNN(cell)
y = layer(x)

### Here's how to use the cell to build a stacked RNN:

cells = [MinimalRNNCell(32), MinimalRNNCell(64)]
x = keras.Input((None, 5))
layer = RNN(cells)
y = layer(x)
```

[source] ##### SimpleRNN

```
keras.layers.SimpleRNN(units, activation='tanh', use_bias=True, kernel_initializer='glorot_u
```

Fully-connected RNN where the output is to be fed back to input.

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use (see activations).
- **Default**: hyperbolic tangent (`tanh`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use__bias**: Boolean, whether the layer uses a bias vector.
- **kernel__initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs (see initializers).
- **recurrent__initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state (see initializers).
- **bias__initializer**: Initializer for the bias vector (see initializers).
- **kernel__regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent__regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias__regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity__regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel__constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).
- **recurrent__constraint**: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias__constraint**: Constraint function applied to the bias vector (see constraints).
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent__dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.
- **return__sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence.
- **return__state**: Boolean. Whether to return the last state in addition to the output.
- **go__backwards**: Boolean (default False). If True, process the input sequence backwards and return the reversed sequence.
- **stateful**: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
- **unroll**: Boolean (default False). If True, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short

sequences.

[source] ##### GRU

```
keras.layers.GRU(units, activation='tanh', recurrent_activation='hard_sigmoid', use_bias=Tru
```

Gated Recurrent Unit - Cho et al. 2014.

There are two variants. The default one is based on 1406.1078v3 and has reset gate applied to hidden state before matrix multiplication. The other one is based on original 1406.1078v1 and has the order reversed.

The second variant is compatible with CuDNNGRU (GPU-only) and allows inference on CPU. Thus it has separate biases for `kernel` and `recurrent_kernel`. Use `'reset_after'=True` and `recurrent_activation='sigmoid'`.

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use (see activations).
- **Default**: hyperbolic tangent (`tanh`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **recurrent_activation**: Activation function to use for the recurrent step (see activations).
- **Default**: hard sigmoid (`hard_sigmoid`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs (see initializers).
- **recurrent_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).
- **recurrent_constraint**: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.
- **implementation**: Implementation mode, either 1 or 2. Mode 1 will structure its operations as a larger number of smaller dot products and additions, whereas mode 2 will batch them into fewer, larger operations. These modes will have different performance profiles on different hardware and for different applications.
- **return_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence.
- **return_state**: Boolean. Whether to return the last state in addition to the output.
- **go_backwards**: Boolean (default False). If True, process the input sequence backwards and return the reversed sequence.
- **stateful**: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
- **unroll**: Boolean (default False). If True, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.
- **reset_after**: GRU convention (whether to apply reset gate after or before matrix multiplication). False = "before" (default), True = "after" (CuDNN compatible).

References

- Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation
- On the Properties of Neural Machine Translation: Encoder-Decoder Approaches
- Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling
- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

---

[source] ##### LSTM

```
keras.layers.LSTM(units, activation='tanh', recurrent_activation='hard_sigmoid', use_bias=Tr
```

Long Short-Term Memory layer - Hochreiter 1997.

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use (see activations).

- **Default**: hyperbolic tangent (`tanh`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **recurrent_activation**: Activation function to use for the recurrent step (see activations).
- **Default**: hard sigmoid (`hard_sigmoid`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. (see initializers).
- **recurrent_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **unit_forget_bias**: Boolean. If True, add 1 to the bias of the forget gate at initialization. Setting it to true will also force `bias_initializer="zeros"`. This is recommended in Jozefowicz et al.
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).
- **recurrent_constraint**: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.
- **implementation**: Implementation mode, either 1 or 2. Mode 1 will structure its operations as a larger number of smaller dot products and additions, whereas mode 2 will batch them into fewer, larger operations. These modes will have different performance profiles on different hardware and for different applications.
- **return_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence.
- **return_state**: Boolean. Whether to return the last state in addition to the output.
- **go_backwards**: Boolean (default False). If True, process the input sequence backwards and return the reversed sequence.

- **stateful**: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
- **unroll**: Boolean (default False). If True, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.

References

- Long short-term memory (original 1997 paper)
- Learning to forget: Continual prediction with LSTM
- Supervised sequence labeling with recurrent neural networks
- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

---

[source] ##### ConvLSTM2D

```
keras.layers.ConvLSTM2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=N
```

Convolutional LSTM.

It is similar to an LSTM layer, but the input transformations and recurrent transformations are both convolutional.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size**: An integer or tuple/list of n integers, specifying the dimensions of the convolution window.
- **strides**: An integer or tuple/list of n integers, specifying the strides of the convolution. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, time, ..., channels)` while `channels_first` corresponds to inputs with shape `(batch, time, channels, ...)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: An integer or tuple/list of n integers, specifying the dilation rate to use for dilated convolution. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any `strides` value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) =`

`x`).

- **recurrent_activation**: Activation function to use for the recurrent step (see activations).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. (see initializers).
- **recurrent_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **unit_forget_bias**: Boolean. If True, add 1 to the bias of the forget gate at initialization. Use in combination with `bias_initializer="zeros"`. This is recommended in Jozefowicz et al.
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).
- **recurrent_constraint**: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).
- **return_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence.
- **go_backwards**: Boolean (default False). If True, process the input sequence backwards.
- **stateful**: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.

Input shape

- if data_format='channels_first' 5D tensor with shape: `(samples, time, channels, rows, cols)`
- if data_format='channels_last' 5D tensor with shape: `(samples, time, rows, cols, channels)`

Output shape

- if `return_sequences`
- if data_format='channels_first' 5D tensor with shape: `(samples, time, filters, output_row, output_col)`
- if data_format='channels_last' 5D tensor with shape: `(samples, time, output_row, output_col, filters)`
- else
- if data_format ='channels_first' 4D tensor with shape: `(samples, filters, output_row, output_col)`
- if data_format='channels_last' 4D tensor with shape: `(samples, output_row, output_col, filters)` where o_row and o_col depend on the shape of the filter and the padding

Raises

- **ValueError**: in case of invalid constructor arguments.

References

- Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting The current implementation does not include the feedback loop on the cells output

---

[source] ##### SimpleRNNCell

```
keras.layers.SimpleRNNCell(units, activation='tanh', use_bias=True, kernel_initializer='glo
```

Cell class for SimpleRNN.

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use (see activations).
- **Default**: hyperbolic tangent (`tanh`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs (see initializers).
- **recurrent_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).

- **recurrent_constraint**: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.

[source] ##### GRUCell

```
keras.layers.GRUCell(units, activation='tanh', recurrent_activation='hard_sigmoid', use_bias
```

Cell class for the GRU layer.

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use (see activations).
- **Default**: hyperbolic tangent (`tanh`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **recurrent_activation**: Activation function to use for the recurrent step (see activations).
- **Default**: hard sigmoid (`hard_sigmoid`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs (see initializers).
- **recurrent_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).
- **recurrent_constraint**: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.

- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.
- **implementation**: Implementation mode, either 1 or 2. Mode 1 will structure its operations as a larger number of smaller dot products and additions, whereas mode 2 will batch them into fewer, larger operations. These modes will have different performance profiles on different hardware and for different applications.
- **reset_after**: GRU convention (whether to apply reset gate after or before matrix multiplication). False = "before" (default), True = "after" (CuDNN compatible).

[source] ##### LSTMCell

```
keras.layers.LSTMCell(units, activation='tanh', recurrent_activation='hard_sigmoid', use_bia
```

Cell class for the LSTM layer.

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use (see activations).
- **Default**: hyperbolic tangent (`tanh`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **recurrent_activation**: Activation function to use for the recurrent step (see activations).
- **Default**: hard sigmoid (`hard_sigmoid`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).x
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs (see initializers).
- **recurrent_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **unit_forget_bias**: Boolean. If True, add 1 to the bias of the forget gate at initialization. Setting it to true will also force `bias_initializer="zeros"`. This is recommended in Jozefowicz et al.
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).

- **recurrent_constraint**: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.
- **implementation**: Implementation mode, either 1 or 2. Mode 1 will structure its operations as a larger number of smaller dot products and additions, whereas mode 2 will batch them into fewer, larger operations. These modes will have different performance profiles on different hardware and for different applications.

[source] ##### CuDNNGRU

```
keras.layers.CuDNNGRU(units, kernel_initializer='glorot_uniform', recurrent_initializer='ort
```

Fast GRU implementation backed by CuDNN.

Can only be run on GPU, with the TensorFlow backend.

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **kernel_initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. (see initializers).
- **recurrent_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).
- **recurrent_constraint**: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).
- **return_sequences**: Boolean. Whether to return the last output. in the output sequence, or the full sequence.

- **return_state**: Boolean. Whether to return the last state in addition to the output.
- **stateful**: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.

[source] ##### CuDNNLSTM

```
keras.layers.CuDNNLSTM(units, kernel_initializer='glorot_uniform', recurrent_initializer='o
```

Fast LSTM implementation backed by CuDNN.

Can only be run on GPU, with the TensorFlow backend.

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **kernel_initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. (see initializers).
- **unit_forget_bias**: Boolean. If True, add 1 to the bias of the forget gate at initialization. Setting it to true will also force `bias_initializer="zeros"`. This is recommended in Jozefowicz et al.
- **recurrent_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).
- **recurrent_constraint**: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).
- **return_sequences**: Boolean. Whether to return the last output. in the output sequence, or the full sequence.
- **return_state**: Boolean. Whether to return the last state in addition to the output.
- **stateful**: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i

in the following batch.

Embedding Layers

[source] ##### Embedding

```
keras.layers.Embedding(input_dim, output_dim, embeddings_initializer='uniform', embeddings_r
```

Turns positive integers (indexes) into dense vectors of fixed size. eg. [[4], [20]] -> [[0.25, 0.1], [0.6, -0.2]]

This layer can only be used as the first layer in a model.

Example

```
model = Sequential()
model.add(Embedding(1000, 64, input_length=10))
### the model will take as input an integer matrix of size (batch, input_length).
### the largest integer (i.e. word index) in the input should be no larger than 999 (vocabu
### now model.output_shape == (None, 10, 64), where None is the batch dimension.

input_array = np.random.randint(1000, size=(32, 10))

model.compile('rmsprop', 'mse')
output_array = model.predict(input_array)
assert output_array.shape == (32, 10, 64)
```

Arguments

- **input_dim**: int > 0. Size of the vocabulary, i.e. maximum integer index + 1.
- **output_dim**: int >= 0. Dimension of the dense embedding.
- **embeddings_initializer**: Initializer for the `embeddings` matrix (see initializers).
- **embeddings_regularizer**: Regularizer function applied to the `embeddings` matrix (see regularizer).
- **embeddings_constraint**: Constraint function applied to the `embeddings` matrix (see constraints).
- **mask_zero**: Whether or not the input value 0 is a special "padding" value that should be masked out. This is useful when using recurrent layers which may take variable length input. If this is `True` then all subsequent layers in the model need to support masking or an exception will be raised. If mask_zero is set to True, as a consequence, index 0 cannot be used in the vocabulary (input_dim should equal size of vocabulary + 1).
- **input_length**: Length of input sequences, when it is constant. This argument is required if you are going to connect `Flatten` then `Dense` layers upstream (without it, the shape of the dense outputs cannot be computed).

Input shape

2D tensor with shape: `(batch_size, sequence_length)`.

Output shape

3D tensor with shape: `(batch_size, sequence_length, output_dim)`.

References

- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

Merge Layers

[source] ##### Add

`keras.layers.Add()`

Layer that adds a list of inputs.

It takes as input a list of tensors, all of the same shape, and returns a single tensor (also of the same shape).

Examples

```
import keras

input1 = keras.layers.Input(shape=(16,))
x1 = keras.layers.Dense(8, activation='relu')(input1)
input2 = keras.layers.Input(shape=(32,))
x2 = keras.layers.Dense(8, activation='relu')(input2)
added = keras.layers.Add()([x1, x2])  # equivalent to added = keras.layers.add([x1, x2])

out = keras.layers.Dense(4)(added)
model = keras.models.Model(inputs=[input1, input2], outputs=out)
```

[source] ##### Subtract

`keras.layers.Subtract()`

Layer that subtracts two inputs.

It takes as input a list of tensors of size 2, both of the same shape, and returns a single tensor, (inputs[0] - inputs[1]), also of the same shape.

Examples

```
import keras

input1 = keras.layers.Input(shape=(16,))
```

```
x1 = keras.layers.Dense(8, activation='relu')(input1)
input2 = keras.layers.Input(shape=(32,))
x2 = keras.layers.Dense(8, activation='relu')(input2)
### Equivalent to subtracted = keras.layers.subtract([x1, x2])
subtracted = keras.layers.Subtract()([x1, x2])

out = keras.layers.Dense(4)(subtracted)
model = keras.models.Model(inputs=[input1, input2], outputs=out)
```

[source] ##### Multiply

`keras.layers.Multiply()`

Layer that multiplies (element-wise) a list of inputs.

It takes as input a list of tensors, all of the same shape, and returns a single tensor (also of the same shape).

[source] ##### Average

`keras.layers.Average()`

Layer that averages a list of inputs.

It takes as input a list of tensors, all of the same shape, and returns a single tensor (also of the same shape).

[source] ##### Maximum

`keras.layers.Maximum()`

Layer that computes the maximum (element-wise) a list of inputs.

It takes as input a list of tensors, all of the same shape, and returns a single tensor (also of the same shape).

[source] ##### Concatenate

`keras.layers.Concatenate(axis=-1)`

Layer that concatenates a list of inputs.

It takes as input a list of tensors, all of the same shape except for the concatenation axis, and returns a single tensor, the concatenation of all inputs.

Arguments

- **axis**: Axis along which to concatenate.
- ___**kwargs___: standard layer keyword arguments.

[source] ##### Dot

`keras.layers.Dot(axes, normalize=False)`

Layer that computes a dot product between samples in two tensors.

E.g. if applied to two tensors `a` and `b` of shape `(batch_size, n)`, the output will be a tensor of shape `(batch_size, 1)` where each entry `i` will be the dot product between `a[i]` and `b[i]`.

Arguments

- **axes**: Integer or tuple of integers, axis or axes along which to take the dot product.
- **normalize**: Whether to L2-normalize samples along the dot product axis before taking the dot product. If set to True, then the output of the dot product is the cosine proximity between the two samples.
- ___**kwargs___: Standard layer keyword arguments.

---

add

`keras.layers.add(inputs)`

Functional interface to the `Add` layer.

Arguments

- **inputs**: A list of input tensors (at least 2).
- ___**kwargs___: Standard layer keyword arguments.

Returns

A tensor, the sum of the inputs.

Examples

```
import keras

input1 = keras.layers.Input(shape=(16,))
x1 = keras.layers.Dense(8, activation='relu')(input1)
input2 = keras.layers.Input(shape=(32,))
x2 = keras.layers.Dense(8, activation='relu')(input2)
added = keras.layers.add([x1, x2])

out = keras.layers.Dense(4)(added)
model = keras.models.Model(inputs=[input1, input2], outputs=out)
```

---

subtract

```
keras.layers.subtract(inputs)
```

Functional interface to the `Subtract` layer.

Arguments

- **inputs**: A list of input tensors (exactly 2).
- ___**kwargs___: Standard layer keyword arguments.

Returns

A tensor, the difference of the inputs.

Examples

```
import keras

input1 = keras.layers.Input(shape=(16,))
x1 = keras.layers.Dense(8, activation='relu')(input1)
input2 = keras.layers.Input(shape=(32,))
x2 = keras.layers.Dense(8, activation='relu')(input2)
subtracted = keras.layers.subtract([x1, x2])

out = keras.layers.Dense(4)(subtracted)
model = keras.models.Model(inputs=[input1, input2], outputs=out)
```

---

multiply

```
keras.layers.multiply(inputs)
```

Functional interface to the `Multiply` layer.

Arguments

- **inputs**: A list of input tensors (at least 2).
- ___**kwargs___: Standard layer keyword arguments.

Returns

A tensor, the element-wise product of the inputs.

---

average

```
keras.layers.average(inputs)
```

Functional interface to the `Average` layer.

Arguments

- **inputs**: A list of input tensors (at least 2).
- ___**kwargs___: Standard layer keyword arguments.

Returns

A tensor, the average of the inputs.

---

maximum

`keras.layers.maximum(inputs)`

Functional interface to the `Maximum` layer.

Arguments

- **inputs**: A list of input tensors (at least 2).
- ___**kwargs___: Standard layer keyword arguments.

Returns

A tensor, the element-wise maximum of the inputs.

---

concatenate

`keras.layers.concatenate(inputs, axis=-1)`

Functional interface to the `Concatenate` layer.

Arguments

- **inputs**: A list of input tensors (at least 2).
- **axis**: Concatenation axis.
- ___**kwargs___: Standard layer keyword arguments.

Returns

A tensor, the concatenation of the inputs alongside axis `axis`.

---

dot

`keras.layers.dot(inputs, axes, normalize=False)`

Functional interface to the `Dot` layer.

Arguments

- **inputs**: A list of input tensors (at least 2).

- **axes**: Integer or tuple of integers, axis or axes along which to take the dot product.
- **normalize**: Whether to L2-normalize samples along the dot product axis before taking the dot product. If set to True, then the output of the dot product is the cosine proximity between the two samples.
- ___**kwargs___: Standard layer keyword arguments.

Returns

A tensor, the dot product of the samples from the inputs.

Advanced Activations Layers

[source] ##### PReLU

```
keras.layers.PReLU(alpha_initializer='zeros', alpha_regularizer=None, alpha_constraint=None
```

Parametric Rectified Linear Unit.

It follows: `f(x) = alpha * x for x < 0`, `f(x) = x for x >= 0`, where `alpha` is a learned array with the same shape as x.

Input shape

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

Output shape

Same shape as the input.

Arguments

- **alpha_initializer**: initializer function for the weights.
- **alpha_regularizer**: regularizer for the weights.
- **alpha_constraint**: constraint for the weights.
- **shared_axes**: the axes along which to share learnable parameters for the activation function. For example, if the incoming feature maps are from a 2D convolution with output shape `(batch, height, width, channels)`, and you wish to share parameters across space so that each filter only has one set of parameters, set `shared_axes=[1, 2]`.

References

- Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

---

[source] ##### ELU

```
keras.layers.ELU(alpha=1.0)
```

Exponential Linear Unit.

It follows: `f(x) =  alpha * (exp(x) - 1.) for x < 0`, `f(x) = x for x >= 0`.

Input shape

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

Output shape

Same shape as the input.

Arguments

- **alpha**: scale for the negative factor.

References

- Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)

[source] ##### ThresholdedReLU

```
keras.layers.ThresholdedReLU(theta=1.0)
```

Thresholded Rectified Linear Unit.

It follows: `f(x) = x for x > theta`, `f(x) = 0 otherwise`.

Input shape

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

Output shape

Same shape as the input.

Arguments

- **theta**: float >= 0. Threshold location of activation.

References

- Zero-Bias Autoencoders and the Benefits of Co-Adapting Features

---

[source] ##### Softmax

```
keras.layers.Softmax(axis=-1)
```

Softmax activation function.

Input shape

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

Output shape

Same shape as the input.

Arguments

- **axis**: Integer, axis along which the softmax normalization is applied.

[source] ##### LeakyReLU

```
keras.layers.LeakyReLU(alpha=0.3)
```

Leaky version of a Rectified Linear Unit.

It allows a small gradient when the unit is not active: `f(x) = alpha * x for x < 0`, `f(x) = x for x >= 0`.

Input shape

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

Output shape

Same shape as the input.

Arguments

- **alpha**: float >= 0. Negative slope coefficient.

References

- Rectifier Nonlinearities Improve Neural Network Acoustic Models

Normalization Layers

[source] ##### BatchNormalization

```
keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=Tr
```

Batch normalization layer (Ioffe and Szegedy, 2014).

Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.

Arguments

- **axis**: Integer, the axis that should be normalized (typically the features axis). For instance, after a `Conv2D` layer with `data_format="channels_first"`, set `axis=1` in `BatchNormalization`.

- **momentum**: Momentum for the moving mean and the moving variance.
- **epsilon**: Small float added to variance to avoid dividing by zero.
- **center**: If True, add offset of `beta` to normalized tensor. If False, `beta` is ignored.
- **scale**: If True, multiply by `gamma`. If False, `gamma` is not used. When the next layer is linear (also e.g. `nn.relu`), this can be disabled since the scaling will be done by the next layer.
- **beta_initializer**: Initializer for the beta weight.
- **gamma_initializer**: Initializer for the gamma weight.
- **moving_mean_initializer**: Initializer for the moving mean.
- **moving_variance_initializer**: Initializer for the moving variance.
- **beta_regularizer**: Optional regularizer for the beta weight.
- **gamma_regularizer**: Optional regularizer for the gamma weight.
- **beta_constraint**: Optional constraint for the beta weight.
- **gamma_constraint**: Optional constraint for the gamma weight.

Input shape

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

Output shape

Same shape as input.

References

- Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Noise layers

[source] ##### GaussianNoise

`keras.layers.GaussianNoise(stddev)`

Apply additive zero-centered Gaussian noise.

This is useful to mitigate overfitting (you could see it as a form of random data augmentation). Gaussian Noise (GS) is a natural choice as corruption process for real valued inputs.

As it is a regularization layer, it is only active at training time.

Arguments

- **stddev**: float, standard deviation of the noise distribution.

Input shape

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

Output shape

Same shape as input.

[source] ##### GaussianDropout

`keras.layers.GaussianDropout(rate)`

Apply multiplicative 1-centered Gaussian noise.

As it is a regularization layer, it is only active at training time.

Arguments

- **rate**: float, drop probability (as with `Dropout`). The multiplicative noise will have standard deviation `sqrt(rate / (1 - rate))`.

Input shape

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

Output shape

Same shape as input.

References

- Dropout: A Simple Way to Prevent Neural Networks from Overfitting Srivastava, Hinton, et al. 2014

[source] ##### AlphaDropout

`keras.layers.AlphaDropout(rate, noise_shape=None, seed=None)`

Applies Alpha Dropout to the input.

Alpha Dropout is a `Dropout` that keeps mean and variance of inputs to their original values, in order to ensure the self-normalizing property even after this dropout. Alpha Dropout fits well to Scaled Exponential Linear Units by randomly setting activations to the negative saturation value.

Arguments

- **rate**: float, drop probability (as with `Dropout`). The multiplicative noise will have standard deviation `sqrt(rate / (1 - rate))`.
- **seed**: A Python integer to use as random seed.

Input shape

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

Output shape

Same shape as input.

References

- Self-Normalizing Neural Networks

Layer wrappers

[source] ##### TimeDistributed

`keras.layers.TimeDistributed(layer)`

This wrapper applies a layer to every temporal slice of an input.

The input should be at least 3D, and the dimension of index one will be considered to be the temporal dimension.

Consider a batch of 32 samples, where each sample is a sequence of 10 vectors of 16 dimensions. The batch input shape of the layer is then `(32, 10, 16)`, and the `input_shape`, not including the samples dimension, is `(10, 16)`.

You can then use `TimeDistributed` to apply a `Dense` layer to each of the 10 timesteps, independently:

```
### as the first layer in a model
model = Sequential()
model.add(TimeDistributed(Dense(8), input_shape=(10, 16)))
### now model.output_shape == (None, 10, 8)
```

The output will then have shape `(32, 10, 8)`.

In subsequent layers, there is no need for the `input_shape`:

```
model.add(TimeDistributed(Dense(32)))
### now model.output_shape == (None, 10, 32)
```

The output will then have shape `(32, 10, 32)`.

`TimeDistributed` can be used with arbitrary layers, not just `Dense`, for instance with a `Conv2D` layer:

```
model = Sequential()
model.add(TimeDistributed(Conv2D(64, (3, 3)),
                          input_shape=(10, 299, 299, 3)))
```

Arguments

- **layer**: a layer instance.

[source] ##### Bidirectional

`keras.layers.Bidirectional(layer, merge_mode='concat', weights=None)`

Bidirectional wrapper for RNNs.

Arguments

- **layer**: `Recurrent` instance.
- **merge_mode**: Mode by which outputs of the forward and backward RNNs will be combined. One of {'sum', 'mul', 'concat', 'ave', None}. If None, the outputs will not be combined, they will be returned as a list.

Raises

- **ValueError**: In case of invalid `merge_mode` argument.

Examples

```
model = Sequential()
model.add(Bidirectional(LSTM(10, return_sequences=True),
                        input_shape=(5, 10)))
model.add(Bidirectional(LSTM(10)))
model.add(Dense(5))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')
```

Writing your own Keras layers

Writing your own Keras layers

For simple, stateless custom operations, you are probably better off using `layers.core.Lambda` layers. But for any custom operation that has trainable weights, you should implement your own layer.

Here is the skeleton of a Keras layer, **as of Keras 2.0** (if you have an older version, please upgrade). There are only three methods you need to implement:

- `build(input_shape)`: this is where you will define your weights. This method must set `self.built = True`, which can be done by calling `super([Layer], self).build()`.
- `call(x)`: this is where the layer's logic lives. Unless you want your layer to support masking, you only have to care about the first argument passed to `call`: the input tensor.
- `compute_output_shape(input_shape)`: in case your layer modifies the shape of its input, you should specify here the shape transformation logic. This allows Keras to do automatic shape inference.

```
from keras import backend as K
from keras.engine.topology import Layer
import numpy as np

class MyLayer(Layer):
    def __init__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(MyLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        # Create a trainable weight variable for this layer.
        self.kernel = self.add_weight(name='kernel',
                                      shape=(input_shape[1], self.output_dim),
                                      initializer='uniform',
                                      trainable=True)
        super(MyLayer, self).build(input_shape)  # Be sure to call this somewhere!

    def call(self, x):
        return K.dot(x, self.kernel)

    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.output_dim)
```

The existing Keras layers provide examples of how to implement almost anything. Never hesitate to read the source code!

Sequence Preprocessing

[source] ##### TimeseriesGenerator

```
keras.preprocessing.sequence.TimeseriesGenerator(data, targets, length, sampling_rate=1, st
```

Utility class for generating batches of temporal data.

This class takes in a sequence of data-points gathered at equal intervals, along with time series parameters such as stride, length of history, etc., to produce batches for training/validation.

Arguments

- **data**: Indexable generator (such as list or Numpy array) containing consecutive data points (timesteps). The data should be at 2D, and axis 0 is expected to be the time dimension.
- **targets**: Targets corresponding to timesteps in `data`. It should have same length as `data`.
- **length**: Length of the output sequences (in number of timesteps).
- **sampling_rate**: Period between successive individual timesteps within sequences. For rate `r`, timesteps `data[i]`, `data[i-r]`, … `data[i - length]` are used for create a sample sequence.
- **stride**: Period between successive output sequences. For stride `s`, consecutive output samples would be centered around `data[i]`, `data[i+s]`, `data[i+2*s]`, etc. start_index, end_index: Data points earlier than

start_index or later than end_index will not be used in the output sequences. This is useful to reserve part of the data for test or validation.
- **shuffle**: Whether to shuffle output samples, or instead draw them in chronological order.
- **reverse**: Boolean: if true, timesteps in each output sample will be in reverse chronological order.
- **batch_size**: Number of timeseries samples in each batch (except maybe the last one).

Returns

A Sequence instance.

Examples

```
from keras.preprocessing.sequence import TimeseriesGenerator
import numpy as np

data = np.array([[i] for i in range(50)])
targets = np.array([[i] for i in range(50)])

data_gen = TimeseriesGenerator(data, targets,
                               length=10, sampling_rate=2,
                               batch_size=2)
assert len(data_gen) == 20

batch_0 = data_gen[0]
x, y = batch_0
assert np.array_equal(x,
                      np.array([[[0], [2], [4], [6], [8]],
                                [[1], [3], [5], [7], [9]]]))
assert np.array_equal(y,
                      np.array([[10], [11]]))
```

---

pad_sequences

```
pad_sequences(sequences, maxlen=None, dtype='int32', padding='pre', truncating='pre', value=
```

Pads sequences to the same length.

This function transforms a list of num_samples sequences (lists of integers) into a 2D Numpy array of shape (num_samples, num_timesteps). num_timesteps is either the maxlen argument if provided, or the length of the longest sequence otherwise.

Sequences that are shorter than num_timesteps are padded with value at the end.

Sequences longer than `num_timesteps` are truncated so that they fit the desired length. The position where padding or truncation happens is determined by the arguments `padding` and `truncating`, respectively.

Pre-padding is the default.

Arguments

- **sequences**: List of lists, where each element is a sequence.
- **maxlen**: Int, maximum length of all sequences.
- **dtype**: Type of the output sequences.
- **padding**: String, 'pre' or 'post': pad either before or after each sequence.
- **truncating**: String, 'pre' or 'post': remove values from sequences larger than `maxlen`, either at the beginning or at the end of the sequences.
- **value**: Float, padding value.

Returns

- **x**: Numpy array with shape `(len(sequences), maxlen)`

Raises

- **ValueError**: In case of invalid values for `truncating` or `padding`, or in case of invalid shape for a `sequences` entry.

---

skipgrams

```
skipgrams(sequence, vocabulary_size, window_size=4, negative_samples=1.0, shuffle=True, cate
```

Generates skipgram word pairs.

This function transforms a sequence of word indexes (list of integers) into tuples of words of the form:

- (word, word in the same window), with label 1 (positive samples).
- (word, random word from the vocabulary), with label 0 (negative samples).

Read more about Skipgram in this gnomic paper by Mikolov et al.: Efficient Estimation of Word Representations in Vector Space

Arguments

- **sequence**: A word sequence (sentence), encoded as a list of word indices (integers). If using a `sampling_table`, word indices are expected to match the rank of the words in a reference dataset (e.g. 10 would encode the 10-th most frequently occurring token). Note that index 0 is expected to be a non-word and will be skipped.
- **vocabulary__size**: Int, maximum possible word index + 1

- **window_size**: Int, size of sampling windows (technically half-window). The window of a word `w_i` will be `[i - window_size, i + window_size+1]`.
- **negative_samples**: Float >= 0. 0 for no negative (i.e. random) samples. 1 for same number as positive samples.
- **shuffle**: Whether to shuffle the word couples before returning them.
- **categorical**: bool. if False, labels will be integers (eg. `[0, 1, 1 .. ]`), if `True`, labels will be categorical, e.g. `[[1,0],[0,1],[0,1] .. ]`.
- **sampling_table**: 1D array of size `vocabulary_size` where the entry i encodes the probability to sample a word of rank i.
- **seed**: Random seed.

Returns

couples, labels: where `couples` are int pairs and `labels` are either 0 or 1.

Note

By convention, index 0 in the vocabulary is a non-word and will be skipped.

---

make_sampling_table

```
make_sampling_table(size, sampling_factor=1e-05)
```

Generates a word rank-based probabilistic sampling table.

Used for generating the `sampling_table` argument for `skipgrams`. `sampling_table[i]` is the probability of sampling the word i-th most common word in a dataset (more common words should be sampled less frequently, for balance).

The sampling probabilities are generated according to the sampling distribution used in word2vec:

`p(word) = min(1, sqrt(word_frequency / sampling_factor) / (word_frequency / sampling_factor))`

We assume that the word frequencies follow Zipf's law (s=1) to derive a numerical approximation of frequency(rank):

`frequency(rank) ~ 1/(rank * (log(rank) + gamma) + 1/2 - 1/(12*rank))` where `gamma` is the Euler-Mascheroni constant.

Arguments

- **size**: Int, number of possible words to sample.
- **sampling_factor**: The sampling factor in the word2vec formula.

Returns

A 1D Numpy array of length `size` where the ith entry is the probability that a word of rank i should be sampled.

Text Preprocessing

Text Preprocessing

[source] ##### Tokenizer

```
keras.preprocessing.text.Tokenizer(num_words=None, filters='!"#$%&()*+,-./:;<=>?@[\]^_`{|}~
', lower=True, split=' ', char_level=False, oov_token=None)
```

Text tokenization utility class.

This class allows to vectorize a text corpus, by turning each text into either a sequence of integers (each integer being the index of a token in a dictionary) or into a vector where the coefficient for each token could be binary, based on word count, based on tf-idf…

Arguments

- **num_words**: the maximum number of words to keep, based on word frequency. Only the most common `num_words` words will be kept.
- **filters**: a string where each element is a character that will be filtered from the texts. The default is all punctuation, plus tabs and line breaks, minus the `'` character.
- **lower**: boolean. Whether to convert the texts to lowercase.
- **split**: str. Separator for word splitting.
- **char_level**: if True, every character will be treated as a token.
- **oov_token**: if given, it will be added to word_index and used to replace out-of-vocabulary words during text_to_sequence calls

By default, all punctuation is removed, turning the texts into space-separated sequences of words (words maybe include the `'` character). These sequences are then split into lists of tokens. They will then be indexed or vectorized.

`0` is a reserved index that won't be assigned to any word.

---

hashing_trick

```
hashing_trick(text, n, hash_function=None, filters='!"#$%&()*+,-./:;<=>?@[\]^_`{|}~
', lower=True, split=' ')
```

Converts a text to a sequence of indexes in a fixed-size hashing space.

Arguments

- **text**: Input text (string).
- **n**: Dimension of the hashing space.
- **hash_function**: defaults to python `hash` function, can be 'md5' or any function that takes in input a string and returns a int. Note that 'hash' is

not a stable hashing function, so it is not consistent across different runs, while ‘md5’ is a stable hashing function.
- **filters**: list (or concatenation) of characters to filter out, such as punctuation. Default: ’¡‘#$%&()*+,-./:;<=>?@[]ˆ_‘{|}∼ ’ , includes basic punctuation, tabs, and newlines.
- **lower**: boolean. Whether to set the text to lowercase.
- **split**: str. Separator for word splitting.

Returns

A list of integer word indices (unicity non-guaranteed).

`0` is a reserved index that won’t be assigned to any word.

Two or more words may be assigned to the same index, due to possible collisions by the hashing function. The probability of a collision is in relation to the dimension of the hashing space and the number of distinct objects.

---

one_hot

```
one_hot(text, n, filters='!"#$%&()*+,-./:;<=>?@[\]^_`{|}~
', lower=True, split=' ')
```

One-hot encodes a text into a list of word indexes of size n.

This is a wrapper to the `hashing_trick` function using `hash` as the hashing function; unicity of word to index mapping non-guaranteed.

Arguments

- **text**: Input text (string).
- **n**: int. Size of vocabulary.
- **filters**: list (or concatenation) of characters to filter out, such as punctuation. Default: ’¡‘#$%&()*+,-./:;<=>?@[]ˆ_‘{|}∼ ’ , includes basic punctuation, tabs, and newlines.
- **lower**: boolean. Whether to set the text to lowercase.
- **split**: str. Separator for word splitting.

Returns

List of integers in [1, n]. Each integer encodes a word (unicity non-guaranteed).

---

text_to_word_sequence

```
text_to_word_sequence(text, filters='!"#$%&()*+,-./:;<=>?@[\]^_`{|}~
', lower=True, split=' ')
```

Converts a text to a sequence of words (or tokens).

Arguments

- **text**: Input text (string).
- **filters**: list (or concatenation) of characters to filter out, such as punctuation. Default: ’¡‘#$%&()*+,-./:;<=>?@[]ˆ_‘{|}~ ’ , includes basic punctuation, tabs, and newlines.
- **lower**: boolean. Whether to convert the input to lowercase.
- **split**: str. Separator for word splitting.

Returns

A list of words (or tokens).

Image Preprocessing

Image Preprocessing

[source] #### ImageDataGenerator class

```
keras.preprocessing.image.ImageDataGenerator(featurewise_center=False, samplewise_center=Fal
```

Generate batches of tensor image data with real-time data augmentation. The data will be looped over (in batches).

Arguments

- **featurewise_center**: Boolean. Set input mean to 0 over the dataset, feature-wise.
- **samplewise_center**: Boolean. Set each sample mean to 0.
- **featurewise_std_normalization**: Boolean. Divide inputs by std of the dataset, feature-wise.
- **samplewise_std_normalization**: Boolean. Divide each input by its std.
- **zca_epsilon**: epsilon for ZCA whitening. Default is 1e-6.
- **zca_whitening**: Boolean. Apply ZCA whitening.
- **rotation_range**: Int. Degree range for random rotations.
- **width_shift_range**: Float, 1-D array-like or int
- **float**: fraction of total width, if < 1, or pixels if >= 1. 1-D array-like: random elements from the array.
- **int**: integer number of pixels from interval `(-width_shift_range, +width_shift_range)` With `width_shift_range=2` possible values are integers `[-1, 0, +1]`, same as with `width_shift_range=[-1, 0, +1]`, while with `width_shift_range=1.0` possible values are floats in the interval [-1.0, +1.0).
- **shear_range**: Float. Shear Intensity (Shear angle in counter-clockwise direction in degrees)

- **zoom_range**: Float or [lower, upper]. Range for random zoom. If a float, `[lower, upper] = [1-zoom_range, 1+zoom_range]`.
- **channel_shift_range**: Float. Range for random channel shifts.
- **fill_mode**: One of {"constant", "nearest", "reflect" or "wrap"}. Default is 'nearest'. Points outside the boundaries of the input are filled according to the given mode:
- **'constant'**: kkkkkkkk|abcd|kkkkkkkk (cval=k)
- **'nearest'**: aaaaaaaa|abcd|dddddddd
- **'reflect'**: abcddcba|abcd|dcbaabcd
- **'wrap'**: abcdabcd|abcd|abcdabcd
- **cval**: Float or Int. Value used for points outside the boundaries when `fill_mode = "constant"`.
- **horizontal_flip**: Boolean. Randomly flip inputs horizontally.
- **vertical_flip**: Boolean. Randomly flip inputs vertically.
- **rescale**: rescaling factor. Defaults to None. If None or 0, no rescaling is applied, otherwise we multiply the data by the value provided (before applying any other transformation).
- **preprocessing_function**: function that will be implied on each input. The function will run after the image is resized and augmented. The function should take one argument: one image (Numpy tensor with rank 3), and should output a Numpy tensor with the same shape.
- **data_format**: Image data format, either "channels_first" or "channels_last". "channels_last" mode means that the images should have shape `(samples, height, width, channels)`, "channels_first" mode means that the images should have shape `(samples, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **validation_split**: Float. Fraction of images reserved for validation (strictly between 0 and 1).

Examples

Example of using `.flow(x, y)`:

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
y_train = np_utils.to_categorical(y_train, num_classes)
y_test = np_utils.to_categorical(y_test, num_classes)

datagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True)
```

```
### compute quantities required for featurewise normalization
### (std, mean, and principal components if ZCA whitening is applied)
datagen.fit(x_train)

### fits the model on batches with real-time data augmentation:
model.fit_generator(datagen.flow(x_train, y_train, batch_size=32),
                    steps_per_epoch=len(x_train) / 32, epochs=epochs)

### here's a more "manual" example
for e in range(epochs):
    print('Epoch', e)
    batches = 0
    for x_batch, y_batch in datagen.flow(x_train, y_train, batch_size=32):
        model.fit(x_batch, y_batch)
        batches += 1
        if batches >= len(x_train) / 32:
            # we need to break the loop by hand because
            # the generator loops indefinitely
            break
```

Example of using `.flow_from_directory(directory)`:

```
train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        'data/train',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        'data/validation',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

model.fit_generator(
        train_generator,
        steps_per_epoch=2000,
        epochs=50,
        validation_data=validation_generator,
        validation_steps=800)
```

Example of transforming images and masks together.

```
### we create two instances with the same arguments
data_gen_args = dict(featurewise_center=True,
                     featurewise_std_normalization=True,
                     rotation_range=90.,
                     width_shift_range=0.1,
                     height_shift_range=0.1,
                     zoom_range=0.2)
image_datagen = ImageDataGenerator(**data_gen_args)
mask_datagen = ImageDataGenerator(**data_gen_args)

### Provide the same seed and keyword arguments to the fit and flow methods
seed = 1
image_datagen.fit(images, augment=True, seed=seed)
mask_datagen.fit(masks, augment=True, seed=seed)

image_generator = image_datagen.flow_from_directory(
    'data/images',
    class_mode=None,
    seed=seed)

mask_generator = mask_datagen.flow_from_directory(
    'data/masks',
    class_mode=None,
    seed=seed)

### combine generators into one which yields image and masks
train_generator = zip(image_generator, mask_generator)

model.fit_generator(
    train_generator,
    steps_per_epoch=2000,
    epochs=50)
```

flow

```
flow(x, y=None, batch_size=32, shuffle=True, seed=None, save_to_dir=None, save_prefix='', sa
```

Takes numpy data & label arrays, and generates batches of augmented data.

Arguments

- **x**: Input data. Numpy array of rank 4 or a tuple. If tuple, the first element should contain the images and the second element another numpy array or a list of numpy arrays that gets passed to the output without any modifications. Can be used to feed the model miscellaneous data along with the images.

In case of grayscale data, the channels axis of the image array should have value 1, and in case of RGB data, it should have value 3. - **y**: Labels. - **batch_size**: Int (default: 32). - **shuffle**: Boolean (default: True). - **seed**: Int (default: None). - **save_to_dir**: None or str (default: None). This allows you to optionally specify a directory to which to save the augmented pictures being generated (useful for visualizing what you are doing). - **save_prefix**: Str (default: `''`). Prefix to use for filenames of saved pictures (only relevant if `save_to_dir` is set). - **save_format**: one of "png", "jpeg" (only relevant if `save_to_dir` is set). Default: "png". - **subset**: Subset of data (`"training"` or `"validation"`) if `validation_split` is set in `ImageDataGenerator`.

Returns

An `Iterator` yielding tuples of `(x, y)` where `x` is a numpy array of image data (in the case of a single image input) or a list of numpy arrays (in the case with additional inputs) and `y` is a numpy array of corresponding labels.

random_transform

```
random_transform(x, seed=None)
```

Randomly augments a single image tensor.

Arguments

- **x**: 3D tensor, single image.
- **seed**: Random seed.

Returns

A randomly transformed version of the input (same shape).

Parameters common to all Keras optimizers

The parameters `clipnorm` and `clipvalue` can be used with all optimizers to control gradient clipping:

```
from keras import optimizers

### All parameter gradients will be clipped to
### a maximum norm of 1.
sgd = optimizers.SGD(lr=0.01, clipnorm=1.)
```

```
from keras import optimizers

### All parameter gradients will be clipped to
### a maximum value of 0.5 and
### a minimum value of -0.5.
sgd = optimizers.SGD(lr=0.01, clipvalue=0.5)
```

[source] ##### Adamax

`keras.optimizers.Adamax(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0)`

Adamax optimizer from Adam paper's Section 7.

It is a variant of Adam based on the infinity norm. Default parameters follow those provided in the paper.

Arguments

- **lr**: float >= 0. Learning rate.
- **beta_1/beta_2**: floats, 0 < beta < 1. Generally close to 1.
- **epsilon**: float >= 0. Fuzz factor. If `None`, defaults to `K.epsilon()`.
- **decay**: float >= 0. Learning rate decay over each update.

References

- Adam - A Method for Stochastic Optimization

[source] ##### Nadam

`keras.optimizers.Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=None, schedule_decay=0.00`

Nesterov Adam optimizer.

Much like Adam is essentially RMSprop with momentum, Nadam is Adam RMSprop with Nesterov momentum.

Default parameters follow those provided in the paper. It is recommended to leave the parameters of this optimizer at their default values.

Arguments

- **lr**: float >= 0. Learning rate.
- **beta_1/beta_2**: floats, 0 < beta < 1. Generally close to 1.
- **epsilon**: float >= 0. Fuzz factor. If `None`, defaults to `K.epsilon()`.

References

- Nadam report
- On the importance of initialization and momentum in deep learning

---

[source] ##### TFOptimizer

`keras.optimizers.TFOptimizer(optimizer)`

Wrapper class for native TensorFlow optimizers.

[source] ##### Adam

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=F
```

Adam optimizer.

Default parameters follow those provided in the original paper.

Arguments

- **lr**: float >= 0. Learning rate.
- **beta_1**: float, 0 < beta < 1. Generally close to 1.
- **beta_2**: float, 0 < beta < 1. Generally close to 1.
- **epsilon**: float >= 0. Fuzz factor. If `None`, defaults to `K.epsilon()`.
- **decay**: float >= 0. Learning rate decay over each update.
- **amsgrad**: boolean. Whether to apply the AMSGrad variant of this algorithm from the paper "On the Convergence of Adam and Beyond".

References

- Adam - A Method for Stochastic Optimization
- On the Convergence of Adam and Beyond

[source] ##### SGD

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

Stochastic gradient descent optimizer.

Includes support for momentum, learning rate decay, and Nesterov momentum.

Arguments

- **lr**: float >= 0. Learning rate.
- **momentum**: float >= 0. Parameter that accelerates SGD in the relevant direction and dampens oscillations.
- **decay**: float >= 0. Learning rate decay over each update.
- **nesterov**: boolean. Whether to apply Nesterov momentum.

[source] ##### Adadelta

```
keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=None, decay=0.0)
```

Adadelta optimizer.

It is recommended to leave the parameters of this optimizer at their default values.

Arguments

- **lr**: float >= 0. Learning rate. It is recommended to leave it at the default value.
- **rho**: float >= 0.
- **epsilon**: float >= 0. Fuzz factor. If `None`, defaults to `K.epsilon()`.
- **decay**: float >= 0. Learning rate decay over each update.

References

- Adadelta - an adaptive learning rate method

[source] ##### Adagrad

```
keras.optimizers.Adagrad(lr=0.01, epsilon=None, decay=0.0)
```

Adagrad optimizer.

It is recommended to leave the parameters of this optimizer at their default values.

Arguments

- **lr**: float >= 0. Learning rate.
- **epsilon**: float >= 0. If `None`, defaults to `K.epsilon()`.
- **decay**: float >= 0. Learning rate decay over each update.

References

- Adaptive Subgradient Methods for Online Learning and Stochastic Optimization

---

[source] ##### RMSprop

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
```

RMSProp optimizer.

It is recommended to leave the parameters of this optimizer at their default values (except the learning rate, which can be freely tuned).

This optimizer is usually a good choice for recurrent neural networks.

Arguments

- **lr**: float >= 0. Learning rate.
- **rho**: float >= 0.
- **epsilon**: float >= 0. Fuzz factor. If `None`, defaults to `K.epsilon()`.
- **decay**: float >= 0. Learning rate decay over each update.

References

- rmsprop: Divide the gradient by a running average of its recent magnitude

Activations

Usage of activations

Activations can either be used through an `Activation` layer, or through the `activation` argument supported by all forward layers:

```
from keras.layers import Activation, Dense

model.add(Dense(64))
model.add(Activation('tanh'))
```

This is equivalent to:

```
model.add(Dense(64, activation='tanh'))
```

You can also pass an element-wise TensorFlow/Theano/CNTK function as an activation:

```
from keras import backend as K

model.add(Dense(64, activation=K.tanh))
```

Available activations

softmax

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

Softmax activation function.

Arguments

- **x**: Input tensor.
- **axis**: Integer, axis along which the softmax normalization is applied.

Returns

Tensor, output of softmax transformation.

Raises

- **ValueError**: In case `dim(x) == 1`.

---

elu

```
elu(x, alpha=1.0)
```

Exponential linear unit.

Arguments

- **x**: Input tensor.
- **alpha**: A scalar, slope of negative section.

Returns

The exponential linear activation: `x` if `x > 0` and `alpha * (exp(x)-1)` if `x < 0`.

References

- Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)

---

selu

`selu(x)`

Scaled Exponential Linear Unit (SELU).

SELU is equal to: `scale * elu(x, alpha)`, where alpha and scale are predefined constants. The values of `alpha` and `scale` are chosen so that the mean and variance of the inputs are preserved between two consecutive layers as long as the weights are initialized correctly (see `lecun_normal` initialization) and the number of inputs is "large enough" (see references for more information).

Arguments

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

Returns

The scaled exponential unit activation: `scale * elu(x, alpha)`.

Note

- To be used together with the initialization "lecun_normal".
- To be used together with the dropout variant "AlphaDropout".

References

- Self-Normalizing Neural Networks

---

softplus

`softplus(x)`

Softplus activation function.

Arguments

- **x**: Input tensor.

Returns

The softplus activation: `log(exp(x) + 1`.

---

softsign

`softsign(x)`

Softsign activation function.

Arguments

- **x**: Input tensor.

Returns

The softplus activation: `x / (abs(x) + 1)`.

---

relu

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

Rectified Linear Unit.

Arguments

- **x**: Input tensor.
- **alpha**: Slope of the negative part. Defaults to zero.
- **max_value**: Maximum value for the output.

Returns

The (leaky) rectified linear unit activation: `x` if `x > 0`, `alpha * x` if `x < 0`. If `max_value` is defined, the result is truncated to this value.

---

tanh

`tanh(x)`

Hyperbolic tangent activation function.

---

sigmoid

`sigmoid(x)`

Sigmoid activation function.

---

hard_sigmoid

`hard_sigmoid(x)`

Hard sigmoid activation function.

Faster to compute than sigmoid activation.

Arguments

- `x`: Input tensor.

Returns

Hard sigmoid activation:

- `0` if `x < -2.5`
- `1` if `x > 2.5`
- `0.2 * x + 0.5` if `-2.5 <= x <= 2.5`.

---

linear

`linear(x)`

Linear (i.e. identity) activation function.

On "Advanced Activations"

Activations that are more complex than a simple TensorFlow/Theano/CNTK function (eg. learnable activations, which maintain a state) are available as Advanced Activation layers, and can be found in the module `keras.layers.advanced_activations`. These include `PReLU` and `LeakyReLU`.

Callbacks

Usage of callbacks

A callback is a set of functions to be applied at given stages of the training procedure. You can use callbacks to get a view on internal states and statistics of the model during training. You can pass a list of callbacks (as the keyword argument `callbacks`) to the `.fit()` method of the `Sequential` or `Model` classes. The relevant methods of the callbacks will then be called at each stage of the training.

[source] ##### BaseLogger

```
keras.callbacks.BaseLogger(stateful_metrics=None)
```

Callback that accumulates epoch averages of metrics.

This callback is automatically applied to every Keras model.

Arguments

- **stateful_metrics**: Iterable of string names of metrics that should *not* be averaged over an epoch. Metrics in this list will be logged as-is in `on_epoch_end`. All others will be averaged in `on_epoch_end`.

[source] ##### TerminateOnNaN

```
keras.callbacks.TerminateOnNaN()
```

Callback that terminates training when a NaN loss is encountered.

[source] ##### EarlyStopping

```
keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=0, verbose=0, mode=
```

Stop training when a monitored quantity has stopped improving.

Arguments

- **monitor**: quantity to be monitored.
- **min_delta**: minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement.
- **patience**: number of epochs with no improvement after which training will be stopped.
- **verbose**: verbosity mode.
- **mode**: one of {auto, min, max}. In `min` mode, training will stop when the quantity monitored has stopped decreasing; in `max` mode it will stop when the quantity monitored has stopped increasing; in `auto` mode, the direction is automatically inferred from the name of the monitored quantity.
- **baseline**: Baseline value for the monitored quantity to reach. Training will stop if the model doesn't show improvement over the baseline.

[source] ##### RemoteMonitor

```
keras.callbacks.RemoteMonitor(root='http://localhost:9000', path='/publish/epoch/end/', fiel
```

Callback used to stream events to a server.

Requires the `requests` library. Events are sent to `root + '/publish/epoch/end/'` by default. Calls are HTTP POST, with a `data` argument which is a JSON-encoded dictionary of event data. If send_as_json is set to True, the content type of the request will be application/json. Otherwise the serialized JSON will be send within a form

Arguments

- **root**: String; root url of the target server.
- **path**: String; path relative to `root` to which the events will be sent.
- **field**: String; JSON field under which the data will be stored. The field is used only if the payload is sent within a form (i.e. send_as_json is set to False).
- **headers**: Dictionary; optional custom HTTP headers.
- **send_as_json**: Boolean; whether the request should be send as application/json.

[source] ##### LearningRateScheduler

```
keras.callbacks.LearningRateScheduler(schedule, verbose=0)
```

Learning rate scheduler.

Arguments

- **schedule**: a function that takes an epoch index as input (integer, indexed from 0) and current learning rate and returns a new learning rate as output (float).
- **verbose**: int. 0: quiet, 1: update messages.

[source] ##### TensorBoard

```
keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=0, batch_size=32, write_graph=T
```

TensorBoard basic visualizations.

TensorBoard is a visualization tool provided with TensorFlow.

This callback writes a log for TensorBoard, which allows you to visualize dynamic graphs of your training and test metrics, as well as activation histograms for the different layers in your model.

If you have installed TensorFlow with pip, you should be able to launch TensorBoard from the command line:

```
tensorboard --logdir=/full_path_to_your_logs
```

When using a backend other than TensorFlow, TensorBoard will still work (if you have TensorFlow installed), but the only feature available will be the display of the losses and metrics plots.

Arguments

- **log_dir**: the path of the directory where to save the log files to be parsed by TensorBoard.
- **histogram_freq**: frequency (in epochs) at which to compute activation and weight histograms for the layers of the model. If set to 0, histograms won't be computed. Validation data (or split) must be specified for histogram visualizations.
- **write_graph**: whether to visualize the graph in TensorBoard. The log file can become quite large when write_graph is set to True.
- **write_grads**: whether to visualize gradient histograms in TensorBoard. `histogram_freq` must be greater than 0.
- **batch_size**: size of batch of inputs to feed to the network for histograms computation.
- **write_images**: whether to write model weights to visualize as image in TensorBoard.
- **embeddings_freq**: frequency (in epochs) at which selected embedding layers will be saved.
- **embeddings_layer_names**: a list of names of layers to keep eye on. If None or empty list all the embedding layer will be watched.
- **embeddings_metadata**: a dictionary which maps layer name to a file name in which metadata for this embedding layer is saved. See the details about metadata files format. In case if the same metadata file is used for all embedding layers, string can be passed.

[source] ##### ReduceLROnPlateau

```
keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=10, verbose=0, mo
```

Reduce learning rate when a metric has stopped improving.

Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This callback monitors a quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.

Example

```
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                              patience=5, min_lr=0.001)
model.fit(X_train, Y_train, callbacks=[reduce_lr])
```

Arguments

- **monitor**: quantity to be monitored.

- **factor**: factor by which the learning rate will be reduced. new_lr = lr * factor
- **patience**: number of epochs with no improvement after which learning rate will be reduced.
- **verbose**: int. 0: quiet, 1: update messages.
- **mode**: one of {auto, min, max}. In `min` mode, lr will be reduced when the quantity monitored has stopped decreasing; in `max` mode it will be reduced when the quantity monitored has stopped increasing; in `auto` mode, the direction is automatically inferred from the name of the monitored quantity.
- **min_delta**: threshold for measuring the new optimum, to only focus on significant changes.
- **cooldown**: number of epochs to wait before resuming normal operation after lr has been reduced.
- **min_lr**: lower bound on the learning rate.

[source] ##### Callback

```
keras.callbacks.Callback()
```

Abstract base class used to build new callbacks.

Properties

- **params**: dict. Training parameters (eg. verbosity, batch size, number of epochs…).
- **model**: instance of `keras.models.Model`. Reference of the model being trained.

The `logs` dictionary that callback methods take as argument will contain keys for quantities relevant to the current batch or epoch.

Currently, the `.fit()` method of the `Sequential` model class will include the following quantities in the `logs` that it passes to its callbacks:

- **on_epoch_end**: logs include `acc` and `loss`, and optionally include `val_loss` (if validation is enabled in `fit`), and `val_acc` (if validation and accuracy monitoring are enabled).
- **on_batch_begin**: logs include `size`, the number of samples in the current batch.
- **on_batch_end**: logs include `loss`, and optionally `acc` (if accuracy monitoring is enabled).

[source] ##### CSVLogger

```
keras.callbacks.CSVLogger(filename, separator=',', append=False)
```

Callback that streams epoch results to a csv file.

Supports all values that can be represented as a string, including 1D iterables such as np.ndarray.

Example

```
csv_logger = CSVLogger('training.log')
model.fit(X_train, Y_train, callbacks=[csv_logger])
```

Arguments

- **filename**: filename of the csv file, e.g. 'run/log.csv'.
- **separator**: string used to separate elements in the csv file.
- **append**: True: append if file exists (useful for continuing training). False: overwrite existing file,

---

[source] ##### LambdaCallback

```
keras.callbacks.LambdaCallback(on_epoch_begin=None, on_epoch_end=None, on_batch_begin=None,
```

Callback for creating simple, custom callbacks on-the-fly.

This callback is constructed with anonymous functions that will be called at the appropriate time. Note that the callbacks expects positional arguments, as:

- `on_epoch_begin` and `on_epoch_end` expect two positional arguments: `epoch`, `logs`
- `on_batch_begin` and `on_batch_end` expect two positional arguments: `batch`, `logs`
- `on_train_begin` and `on_train_end` expect one positional argument: `logs`

Arguments

- **on_epoch_begin**: called at the beginning of every epoch.
- **on_epoch_end**: called at the end of every epoch.
- **on_batch_begin**: called at the beginning of every batch.
- **on_batch_end**: called at the end of every batch.
- **on_train_begin**: called at the beginning of model training.
- **on_train_end**: called at the end of model training.

Example

```
### Print the batch number at the beginning of every batch.
batch_print_callback = LambdaCallback(
    on_batch_begin=lambda batch,logs: print(batch))

### Stream the epoch loss to a file in JSON format. The file content
### is not well-formed JSON but rather has a JSON object per line.
import json
json_log = open('loss_log.json', mode='wt', buffering=1)
json_logging_callback = LambdaCallback(
    on_epoch_end=lambda epoch, logs: json_log.write(
        json.dumps({'epoch': epoch, 'loss': logs['loss']}) + '\n'),
    on_train_end=lambda logs: json_log.close()
)

### Terminate some processes after having finished model training.
processes = ...
cleanup_callback = LambdaCallback(
    on_train_end=lambda logs: [
        p.terminate() for p in processes if p.is_alive()])

model.fit(...,
          callbacks=[batch_print_callback,
                     json_logging_callback,
                     cleanup_callback])
```

[source] ##### ProgbarLogger

`keras.callbacks.ProgbarLogger(count_mode='samples', stateful_metrics=None)`

Callback that prints metrics to stdout.

Arguments

- **count_mode**: One of "steps" or "samples". Whether the progress bar should count samples seen or steps (batches) seen.
- **stateful_metrics**: Iterable of string names of metrics that should *not* be averaged over an epoch. Metrics in this list will be logged as-is. All others will be averaged over time (e.g. loss, etc).

Raises

- **ValueError**: In case of invalid `count_mode`.

[source] ##### History

`keras.callbacks.History()`

Callback that records events into a `History` object.

This callback is automatically applied to every Keras model. The `History` object gets returned by the `fit` method of models.

[source] ##### ModelCheckpoint

`keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0, save_best_only=Fals`

Save the model after every epoch.

`filepath` can contain named formatting options, which will be filled the value of `epoch` and keys in `logs` (passed in `on_epoch_end`).

For example: if `filepath` is `weights.{epoch:02d}-{val_loss:.2f}.hdf5`, then the model checkpoints will be saved with the epoch number and the validation loss in the filename.

Arguments

- **filepath**: string, path to save the model file.
- **monitor**: quantity to monitor.
- **verbose**: verbosity mode, 0 or 1.
- **save_best_only**: if `save_best_only=True`, the latest best model according to the quantity monitored will not be overwritten.
- **mode**: one of {auto, min, max}. If `save_best_only=True`, the decision to overwrite the current save file is made based on either the maximization or the minimization of the monitored quantity. For `val_acc`, this should be `max`, for `val_loss` this should be `min`, etc. In `auto` mode, the direction is automatically inferred from the name of the monitored quantity.
- **save_weights_only**: if True, then only the model's weights will be saved (`model.save_weights(filepath)`), else the full model is saved (`model.save(filepath)`).
- **period**: Interval (number of epochs) between checkpoints.

Create a callback

You can create a custom callback by extending the base class `keras.callbacks.Callback`. A callback has access to its associated model through the class property `self.model`.

Here's a simple example saving a list of losses over each batch during training:

```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))
```

Example: recording loss history

```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))

model = Sequential()
model.add(Dense(10, input_dim=784, kernel_initializer='uniform'))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

history = LossHistory()
model.fit(x_train, y_train, batch_size=128, epochs=20, verbose=0, callbacks=[history])

print(history.losses)
### outputs
'''
[0.66047596406559383, 0.3547245744908703, ..., 0.25953155204159617, 0.25901699725311789]
'''
```

Example: model checkpoints

```
from keras.callbacks import ModelCheckpoint

model = Sequential()
model.add(Dense(10, input_dim=784, kernel_initializer='uniform'))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

'''
saves the model weights after each epoch if the validation loss decreased
'''
checkpointer = ModelCheckpoint(filepath='/tmp/weights.hdf5', verbose=1, save_best_only=True
model.fit(x_train, y_train, batch_size=128, epochs=20, verbose=0, validation_data=(X_test, 
```

Datasets

Datasets

CIFAR10 small image classification

Dataset of 50,000 32x32 color training images, labeled over 10 categories, and 10,000 test images.

Usage:

```
from keras.datasets import cifar10

(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

- **Returns:**
  - 2 tuples:
    - **x_train, x_test**: uint8 array of RGB image data with shape (num_samples, 3, 32, 32).
    - **y_train, y_test**: uint8 array of category labels (integers in range 0-9) with shape (num_samples,).

CIFAR100 small image classification

Dataset of 50,000 32x32 color training images, labeled over 100 categories, and 10,000 test images.

Usage:

```
from keras.datasets import cifar100

(x_train, y_train), (x_test, y_test) = cifar100.load_data(label_mode='fine')
```

- **Returns:**
  - 2 tuples:
    - **x_train, x_test**: uint8 array of RGB image data with shape (num_samples, 3, 32, 32).
    - **y_train, y_test**: uint8 array of category labels with shape (num_samples,).
- **Arguments:**
  - **label_mode**: "fine" or "coarse".

IMDB Movie reviews sentiment classification

Dataset of 25,000 movies reviews from IMDB, labeled by sentiment (positive/negative). Reviews have been preprocessed, and each review is encoded as a sequence of word indexes (integers). For convenience, words are indexed by overall frequency in the dataset, so that for instance the integer "3" encodes the 3rd most frequent word in the data. This allows for quick filtering operations such as: "only consider the top 10,000 most common words, but eliminate the top 20 most common words".

As a convention, “0” does not stand for a specific word, but instead is used to encode any unknown word.

Usage:

```
from keras.datasets import imdb

(x_train, y_train), (x_test, y_test) = imdb.load_data(path="imdb.npz",
                                                      num_words=None,
                                                      skip_top=0,
                                                      maxlen=None,
                                                      seed=113,
                                                      start_char=1,
                                                      oov_char=2,
                                                      index_from=3)
```

- **Returns:**
  - 2 tuples:
    * **x__train, x__test**: list of sequences, which are lists of indexes (integers). If the num__words argument was specific, the maximum possible index value is num__words-1. If the maxlen argument was specified, the largest possible sequence length is maxlen.
    * **y__train, y__test**: list of integer labels (1 or 0).
- **Arguments:**
  - **path**: if you do not have the data locally (at `'~/.keras/datasets/' + path`), it will be downloaded to this location.
  - **num__words**: integer or None. Top most frequent words to consider. Any less frequent word will appear as `oov_char` value in the sequence data.
  - **skip__top**: integer. Top most frequent words to ignore (they will appear as `oov_char` value in the sequence data).
  - **maxlen**: int. Maximum sequence length. Any longer sequence will be truncated.
  - **seed**: int. Seed for reproducible data shuffling.
  - **start__char**: int. The start of a sequence will be marked with this character. Set to 1 because 0 is usually the padding character.
  - **oov__char**: int. words that were cut out because of the `num_words` or `skip_top` limit will be replaced with this character.
  - **index__from**: int. Index actual words with this index and higher.

Reuters newswire topics classification

Dataset of 11,228 newswires from Reuters, labeled over 46 topics. As with the IMDB dataset, each wire is encoded as a sequence of word indexes (same conventions).

Usage:

```
from keras.datasets import reuters

(x_train, y_train), (x_test, y_test) = reuters.load_data(path="reuters.npz",
                                                         num_words=None,
                                                         skip_top=0,
                                                         maxlen=None,
                                                         test_split=0.2,
                                                         seed=113,
                                                         start_char=1,
                                                         oov_char=2,
                                                         index_from=3)
```

The specifications are the same as that of the IMDB dataset, with the addition of:

- **test_split**: float. Fraction of the dataset to be used as test data.

This dataset also makes available the word index used for encoding the sequences:

```
word_index = reuters.get_word_index(path="reuters_word_index.json")
```

- **Returns:** A dictionary where key are words (str) and values are indexes (integer). eg. `word_index["giraffe"]` might return `1234`.
- **Arguments:**
  - **path**: if you do not have the index file locally (at `'~/.keras/datasets/' + path`), it will be downloaded to this location.

MNIST database of handwritten digits

Dataset of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images.

Usage:

```
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

- **Returns:**

- 2 tuples:
    - **x__train, x__test**: uint8 array of grayscale image data with shape (num_samples, 28, 28).
    - **y__train, y__test**: uint8 array of digit labels (integers in range 0-9) with shape (num_samples,).

- **Arguments:**
  - **path**: if you do not have the index file locally (at `'~/.keras/datasets/' + path`), it will be downloaded to this location.

Fashion-MNIST database of fashion articles

Dataset of 60,000 28x28 grayscale images of 10 fashion categories, along with a test set of 10,000 images. This dataset can be used as a drop-in replacement for MNIST. The class labels are:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Usage:

```
from keras.datasets import fashion_mnist

(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
```

- **Returns:**
  - 2 tuples:
    - **x__train, x__test**: uint8 array of grayscale image data with shape (num_samples, 28, 28).
    - **y__train, y__test**: uint8 array of labels (integers in range 0-9) with shape (num_samples,).

Boston housing price regression dataset

Dataset taken from the StatLib library which is maintained at Carnegie Mellon University.

Samples contain 13 attributes of houses at different locations around the Boston suburbs in the late 1970s. Targets are the median values of the houses at a location (in k$).

Usage:

```
from keras.datasets import boston_housing

(x_train, y_train), (x_test, y_test) = boston_housing.load_data()
```

- **Arguments:**
  - **path**: path where to cache the dataset locally (relative to ~/.keras/datasets).
  - **seed**: Random seed for shuffling the data before computing the test split.
  - **test_split**: fraction of the data to reserve as test set.
- **Returns:** Tuple of Numpy arrays: `(x_train, y_train), (x_test, y_test)`.

Applications

Applications

Keras Applications are deep learning models that are made available alongside pre-trained weights. These models can be used for prediction, feature extraction, and fine-tuning.

Weights are downloaded automatically when instantiating a model. They are stored at `~/.keras/models/`.

Available models

Models for image classification with weights trained on ImageNet:

- Xception
- VGG16
- VGG19
- ResNet50
- InceptionV3
- InceptionResNetV2

- MobileNet
- DenseNet
- NASNet

All of these architectures (except Xception and MobileNet) are compatible with both TensorFlow and Theano, and upon instantiation the models will be built according to the image data format set in your Keras configuration file at `~/.keras/keras.json`. For instance, if you have set `image_data_format=channels_last`, then any model loaded from this repository will get built according to the TensorFlow data format convention, "Height-Width-Depth".

The Xception model is only available for TensorFlow, due to its reliance on `SeparableConvolution` layers. The MobileNet model is only available for TensorFlow, due to its reliance on `DepthwiseConvolution` layers.

Usage examples for image classification models

Classify ImageNet classes with ResNet50

```
from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input, decode_predictions
import numpy as np

model = ResNet50(weights='imagenet')

img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

preds = model.predict(x)
### decode the results into a list of tuples (class, description, probability)
### (one such list for each sample in the batch)
print('Predicted:', decode_predictions(preds, top=3)[0])
### Predicted: [(u'n02504013', u'Indian_elephant', 0.82658225), (u'n01871265', u'tusker', 0
```

Extract features with VGG16

```
from keras.applications.vgg16 import VGG16
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
```

```
import numpy as np

model = VGG16(weights='imagenet', include_top=False)

img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

features = model.predict(x)
```

Extract features from an arbitrary intermediate layer with VGG19

```
from keras.applications.vgg19 import VGG19
from keras.preprocessing import image
from keras.applications.vgg19 import preprocess_input
from keras.models import Model
import numpy as np

base_model = VGG19(weights='imagenet')
model = Model(inputs=base_model.input, outputs=base_model.get_layer('block4_pool').output)

img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

block4_pool_features = model.predict(x)
```

Fine-tune InceptionV3 on a new set of classes

```
from keras.applications.inception_v3 import InceptionV3
from keras.preprocessing import image
from keras.models import Model
from keras.layers import Dense, GlobalAveragePooling2D
from keras import backend as K

### create the base pre-trained model
base_model = InceptionV3(weights='imagenet', include_top=False)

### add a global spatial average pooling layer
x = base_model.output
x = GlobalAveragePooling2D()(x)
```

```
### let's add a fully-connected layer
x = Dense(1024, activation='relu')(x)
### and a logistic layer -- let's say we have 200 classes
predictions = Dense(200, activation='softmax')(x)

### this is the model we will train
model = Model(inputs=base_model.input, outputs=predictions)

### first: train only the top layers (which were randomly initialized)
### i.e. freeze all convolutional InceptionV3 layers
for layer in base_model.layers:
    layer.trainable = False

### compile the model (should be done *after* setting layers to non-trainable)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

### train the model on the new data for a few epochs
model.fit_generator(...)

### at this point, the top layers are well trained and we can start fine-tuning
### convolutional layers from inception V3. We will freeze the bottom N layers
### and train the remaining top layers.

### let's visualize layer names and layer indices to see how many layers
### we should freeze:
for i, layer in enumerate(base_model.layers):
   print(i, layer.name)

### we chose to train the top 2 inception blocks, i.e. we will freeze
### the first 249 layers and unfreeze the rest:
for layer in model.layers[:249]:
   layer.trainable = False
for layer in model.layers[249:]:
   layer.trainable = True

### we need to recompile the model for these modifications to take effect
### we use SGD with a low learning rate
from keras.optimizers import SGD
model.compile(optimizer=SGD(lr=0.0001, momentum=0.9), loss='categorical_crossentropy')

### we train our model again (this time fine-tuning the top 2 inception blocks
### alongside the top Dense layers
model.fit_generator(...)
```

Build InceptionV3 over a custom input tensor

```
from keras.applications.inception_v3 import InceptionV3
from keras.layers import Input

### this could also be the output a different Keras model or layer
input_tensor = Input(shape=(224, 224, 3))  # this assumes K.image_data_format() == 'channels

model = InceptionV3(input_tensor=input_tensor, weights='imagenet', include_top=True)
```

Documentation for individual models

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.715	0.901	138,357,544	23
VGG19	549 MB	0.727	0.910	143,667,240	26
ResNet50	99 MB	0.759	0.929	25,636,712	168
InceptionV3	92 MB	0.788	0.944	23,851,784	159
InceptionResNetV2	215 MB	0.804	0.953	55,873,736	572
MobileNet	17 MB	0.665	0.871	4,253,864	88
DenseNet121	33 MB	0.745	0.918	8,062,504	121
DenseNet169	57 MB	0.759	0.928	14,307,880	169
DenseNet201	80 MB	0.770	0.933	20,242,984	201

The top-1 and top-5 accuracy refers to the model's performance on the ImageNet validation dataset.

Xception

```
keras.applications.xception.Xception(include_top=True, weights='imagenet', input_tensor=None
```

Xception V1 model, with weights pre-trained on ImageNet.

On ImageNet, this model gets to a top-1 validation accuracy of 0.790 and a top-5 validation accuracy of 0.945.

Note that this model is only available for the TensorFlow backend, due to its reliance on `SeparableConvolution` layers. Additionally it only supports the data format `'channels_last'` (height, width, channels).

The default input size for this model is 299x299.

Arguments

- include_top: whether to include the fully-connected layer at the top of the network.
- weights: one of `None` (random initialization) or `'imagenet'` (pre-training on ImageNet).
- input_tensor: optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
- input_shape: optional shape tuple, only to be specified if `include_top` is `False` (otherwise the input shape has to be `(299, 299, 3)`. It should have exactly 3 inputs channels, and width and height should be no smaller than 71. E.g. `(150, 150, 3)` would be one valid value.
- pooling: Optional pooling mode for feature extraction when `include_top` is `False`.
  - `None` means that the output of the model will be the 4D tensor output of the last convolutional layer.
  - `'avg'` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
  - `'max'` means that global max pooling will be applied.
- classes: optional number of classes to classify images into, only to be specified if `include_top` is `True`, and if no `weights` argument is specified.

Returns

A Keras `Model` instance.

References

- Xception: Deep Learning with Depthwise Separable Convolutions

License

These weights are trained by ourselves and are released under the MIT license.

VGG16

```
keras.applications.vgg16.VGG16(include_top=True, weights='imagenet', input_tensor=None, inpu
```

VGG16 model, with weights pre-trained on ImageNet.

This model is available for both the Theano and TensorFlow backend, and can be built both with `'channels_first'` data format (channels, height, width) or `'channels_last'` data format (height, width, channels).

The default input size for this model is 224x224.

Arguments

- include_top: whether to include the 3 fully-connected layers at the top of the network.
- weights: one of `None` (random initialization) or `'imagenet'` (pre-training on ImageNet).
- input_tensor: optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
- input_shape: optional shape tuple, only to be specified if `include_top` is `False` (otherwise the input shape has to be `(224, 224, 3)` (with `'channels_last'` data format) or `(3, 224, 224)` (with `'channels_first'` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 48. E.g. `(200, 200, 3)` would be one valid value.
- pooling: Optional pooling mode for feature extraction when `include_top` is `False`.
  - `None` means that the output of the model will be the 4D tensor output of the last convolutional layer.
  - `'avg'` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
  - `'max'` means that global max pooling will be applied.
- classes: optional number of classes to classify images into, only to be specified if `include_top` is `True`, and if no `weights` argument is specified.

Returns

A Keras `Model` instance.

References

- Very Deep Convolutional Networks for Large-Scale Image Recognition: please cite this paper if you use the VGG models in your work.

License

These weights are ported from the ones released by VGG at Oxford under the Creative Commons Attribution License.

VGG19

```
keras.applications.vgg19.VGG19(include_top=True, weights='imagenet', input_tensor=None, inpu
```

VGG19 model, with weights pre-trained on ImageNet.

This model is available for both the Theano and TensorFlow backend, and can be built both with `'channels_first'` data format (channels, height, width) or `'channels_last'` data format (height, width, channels).

The default input size for this model is 224x224.

Arguments

- include_top: whether to include the 3 fully-connected layers at the top of the network.
- weights: one of `None` (random initialization) or `'imagenet'` (pre-training on ImageNet).
- input_tensor: optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
- input_shape: optional shape tuple, only to be specified if `include_top` is `False` (otherwise the input shape has to be `(224, 224, 3)` (with `'channels_last'` data format) or `(3, 224, 224)` (with `'channels_first'` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 48. E.g. `(200, 200, 3)` would be one valid value.
- pooling: Optional pooling mode for feature extraction when `include_top` is `False`.
  - `None` means that the output of the model will be the 4D tensor output of the last convolutional layer.
  - `'avg'` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
  - `'max'` means that global max pooling will be applied.
- classes: optional number of classes to classify images into, only to be specified if `include_top` is `True`, and if no `weights` argument is specified.

Returns

A Keras `Model` instance.

References

- Very Deep Convolutional Networks for Large-Scale Image Recognition

License

These weights are ported from the ones released by VGG at Oxford under the Creative Commons Attribution License.

ResNet50

```
keras.applications.resnet50.ResNet50(include_top=True, weights='imagenet', input_tensor=None
```

ResNet50 model, with weights pre-trained on ImageNet.

This model is available for both the Theano and TensorFlow backend, and can be built both with `'channels_first'` data format (channels, height, width) or `'channels_last'` data format (height, width, channels).

The default input size for this model is 224x224.

Arguments

- include_top: whether to include the fully-connected layer at the top of the network.
- weights: one of `None` (random initialization) or `'imagenet'` (pre-training on ImageNet).
- input_tensor: optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
- input_shape: optional shape tuple, only to be specified if `include_top` is `False` (otherwise the input shape has to be `(224, 224, 3)` (with `'channels_last'` data format) or `(3, 224, 224)` (with `'channels_first'` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 197. E.g. `(200, 200, 3)` would be one valid value.
- pooling: Optional pooling mode for feature extraction when `include_top` is `False`.
  - `None` means that the output of the model will be the 4D tensor output of the last convolutional layer.
  - `'avg'` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
  - `'max'` means that global max pooling will be applied.
- classes: optional number of classes to classify images into, only to be specified if `include_top` is `True`, and if no `weights` argument is specified.

Returns

A Keras `Model` instance.

References

- Deep Residual Learning for Image Recognition

License

These weights are ported from the ones released by Kaiming He under the MIT license.

InceptionV3

```
keras.applications.inception_v3.InceptionV3(include_top=True, weights='imagenet', input_tens
```

Inception V3 model, with weights pre-trained on ImageNet.

This model is available for both the Theano and TensorFlow backend, and can be built both with `'channels_first'` data format (channels, height, width) or `'channels_last'` data format (height, width, channels).

The default input size for this model is 299x299.

Arguments

- include_top: whether to include the fully-connected layer at the top of the network.
- weights: one of `None` (random initialization) or `'imagenet'` (pre-training on ImageNet).
- input_tensor: optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
- input_shape: optional shape tuple, only to be specified if `include_top` is `False` (otherwise the input shape has to be `(299, 299, 3)` (with `'channels_last'` data format) or `(3, 299, 299)` (with `'channels_first'` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 139. E.g. `(150, 150, 3)` would be one valid value.
- pooling: Optional pooling mode for feature extraction when `include_top` is `False`.
  - `None` means that the output of the model will be the 4D tensor output of the last convolutional layer.
  - `'avg'` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.

    - 'max' means that global max pooling will be applied.
- classes: optional number of classes to classify images into, only to be specified if `include_top` is `True`, and if no `weights` argument is specified.

Returns

A Keras `Model` instance.

References

- Rethinking the Inception Architecture for Computer Vision

License

These weights are released under the Apache License.

InceptionResNetV2

```
keras.applications.inception_resnet_v2.InceptionResNetV2(include_top=True, weights='imagene
```

Inception-ResNet V2 model, with weights pre-trained on ImageNet.

This model is available for Theano, TensorFlow and CNTK backends, and can be built both with `'channels_first'` data format (channels, height, width) or `'channels_last'` data format (height, width, channels).

The default input size for this model is 299x299.

Arguments

- include_top: whether to include the fully-connected layer at the top of the network.
- weights: one of `None` (random initialization) or `'imagenet'` (pre-training on ImageNet).
- input_tensor: optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
- input_shape: optional shape tuple, only to be specified if `include_top` is `False` (otherwise the input shape has to be `(299, 299, 3)` (with `'channels_last'` data format) or `(3, 299, 299)` (with `'channels_first'` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 139. E.g. `(150, 150, 3)` would be one valid value.
- pooling: Optional pooling mode for feature extraction when `include_top` is `False`.

  - `None` means that the output of the model will be the 4D tensor output of the last convolutional layer.
  - `'avg'` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
  - `'max'` means that global max pooling will be applied.
- classes: optional number of classes to classify images into, only to be specified if `include_top` is `True`, and if no `weights` argument is specified.

Returns

A Keras `Model` instance.

References

- Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning

License

These weights are released under the Apache License.

MobileNet

```
keras.applications.mobilenet.MobileNet(input_shape=None, alpha=1.0, depth_multiplier=1, drop
```

MobileNet model, with weights pre-trained on ImageNet.

Note that only TensorFlow is supported for now, therefore it only works with the data format `image_data_format='channels_last'` in your Keras config at `~/.keras/keras.json`. To load a MobileNet model via `load_model`, import the custom objects `relu6` and `DepthwiseConv2D` and pass them to the `custom_objects` parameter.

E.g.

```
model = load_model('mobilenet.h5', custom_objects={
                   'relu6': mobilenet.relu6,
                   'DepthwiseConv2D': mobilenet.DepthwiseConv2D})
```

The default input size for this model is 224x224.

Arguments

- input_shape: optional shape tuple, only to be specified if `include_top` is `False` (otherwise the input shape has to be `(224, 224, 3)` (with `'channels_last'` data format) or `(3, 224, 224)` (with `'channels_first'` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.
- alpha: controls the width of the network.
  - If `alpha` < 1.0, proportionally decreases the number of filters in each layer.
  - If `alpha` > 1.0, proportionally increases the number of filters in each layer.
  - If `alpha` $= 1$, default number of filters from the paper are used at each layer.
- depth_multiplier: depth multiplier for depthwise convolution (also called the resolution multiplier)
- dropout: dropout rate
- include_top: whether to include the fully-connected layer at the top of the network.
- weights: `None` (random initialization) or `'imagenet'` (ImageNet weights)
- input_tensor: optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
- pooling: Optional pooling mode for feature extraction when `include_top` is `False`.
  - `None` means that the output of the model will be the 4D tensor output of the last convolutional layer.
  - `'avg'` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
  - `'max'` means that global max pooling will be applied.
- classes: optional number of classes to classify images into, only to be specified if `include_top` is `True`, and if no `weights` argument is specified.

Returns

A Keras `Model` instance.

References

- MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

License

These weights are released under the Apache License.

DenseNet

```
keras.applications.densenet.DenseNet121(include_top=True, weights='imagenet', input_tensor=N
keras.applications.densenet.DenseNet169(include_top=True, weights='imagenet', input_tensor=N
keras.applications.densenet.DenseNet201(include_top=True, weights='imagenet', input_tensor=N
```

Optionally loads weights pre-trained on ImageNet. Note that when using TensorFlow, for best performance you should set `image_data_format='channels_last'` in your Keras config at ~/.keras/keras.json.

The model and the weights are compatible with TensorFlow, Theano, and CNTK. The data format convention used by the model is the one specified in your Keras config file.

Arguments

- blocks: numbers of building blocks for the four dense layers.
- include_top: whether to include the fully-connected layer at the top of the network.
- weights: one of `None` (random initialization), 'imagenet' (pre-training on ImageNet), or the path to the weights file to be loaded.
- input_tensor: optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
- input_shape: optional shape tuple, only to be specified if `include_top` is False (otherwise the input shape has to be `(224, 224, 3)` (with `channels_last` data format) or `(3, 224, 224)` (with `channels_first` data format). It should have exactly 3 inputs channels.
- pooling: optional pooling mode for feature extraction when `include_top` is `False`.
  - `None` means that the output of the model will be the 4D tensor output of the last convolutional layer.
  - `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
  - `max` means that global max pooling will be applied.
- classes: optional number of classes to classify images into, only to be specified if `include_top` is True, and if no `weights` argument is specified.

Returns

A Keras model instance.

References

- Densely Connected Convolutional Networks (CVPR 2017 Best Paper Award)

License

These weights are released under the BSD 3-clause License.

NASNet

```
keras.applications.nasnet.NASNetLarge(input_shape=None, include_top=True, weights='imagenet'
keras.applications.nasnet.NASNetMobile(input_shape=None, include_top=True, weights='imagenet
```

Neural Architecture Search Network (NASNet) model, with weights pre-trained on ImageNet.

Note that only TensorFlow is supported for now, therefore it only works with the data format `image_data_format='channels_last'` in your Keras config at `~/.keras/keras.json`.

The default input size for the NASNetLarge model is 331x331 and for the NASNetMobile model is 224x224.

Arguments

- input_shape: optional shape tuple, only to be specified if `include_top` is `False` (otherwise the input shape has to be `(224, 224, 3)` (with `'channels_last'` data format) or `(3, 224, 224)` (with `'channels_first'` data format) for NASNetMobile or `(331, 331, 3)` (with `'channels_last'` data format) or `(3, 331, 331)` (with `'channels_first'` data format) for NASNetLarge. It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.
- include_top: whether to include the fully-connected layer at the top of the network.
- weights: `None` (random initialization) or `'imagenet'` (ImageNet weights)
- input_tensor: optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
- pooling: Optional pooling mode for feature extraction when `include_top` is `False`.
  - `None` means that the output of the model will be the 4D tensor output of the last convolutional layer.
  - `'avg'` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
  - `'max'` means that global max pooling will be applied.
- classes: optional number of classes to classify images into, only to be specified if `include_top` is `True`, and if no `weights` argument is specified.

Returns

A Keras `Model` instance.

References

- Learning Transferable Architectures for Scalable Image Recognition

License

These weights are released under the Apache License.

Backend

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 three backend implementations available: the **TensorFlow** backend, the **Theano** backend, and the **CNTK** backend.

- TensorFlow is an open-source symbolic tensor manipulation framework developed by Google.
- Theano is an open-source symbolic tensor manipulation framework developed by LISA Lab at Université de Montréal.
- CNTK is an open-source toolkit for deep learning developed by Microsoft.

In the future, we are likely to add more backend options.

Switching from one backend to another

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

`$HOME/.keras/keras.json`

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

NOTE for Windows Users: Please replace `$HOME` with `%USERPROFILE%`.

The default configuration file looks like this:

```
{
    "image_data_format": "channels_last",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "tensorflow"
}
```

Simply change the field `backend` to `"theano"`, `"tensorflow"`, or `"cntk"`, 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"
Using TensorFlow backend.
```

---

keras.json details

The `keras.json` configuration file contains the following settings:

```
{
    "image_data_format": "channels_last",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "tensorflow"
}
```

You can change these settings by editing `$HOME/.keras/keras.json`.

- `image_data_format`: String, either `"channels_last"` or `"channels_first"`. It specifies which data format convention Keras will follow. (`keras.backend.image_data_format()` returns it.)
- For 2D data (e.g. image), `"channels_last"` assumes `(rows, cols, channels)` while `"channels_first"` assumes `(channels, rows, cols)`.
- For 3D data, `"channels_last"` assumes `(conv_dim1, conv_dim2, conv_dim3, channels)` while `"channels_first"` assumes `(channels, conv_dim1, conv_dim2, conv_dim3)`.
- `epsilon`: Float, a numeric fuzzing constant used to avoid dividing by zero in some operations.
- `floatx`: String, `"float16"`, `"float32"`, or `"float64"`. Default float precision.
- `backend`: String, `"tensorflow"`, `"theano"`, or `"cntk"`.

Using the abstract Keras backend to write new code

If you want the Keras modules you write to be compatible with both Theano (`th`) and TensorFlow (`tf`), 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 `th.tensor.matrix()`, `th.tensor.tensor3()`, etc.

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

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

```
import numpy as np
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:

```
### Initializing Tensors with Random Numbers
b = K.random_uniform_variable(shape=(3, 4), low=0, high=1) # Uniform distribution
c = K.random_normal_variable(shape=(3, 4), mean=0, scale=1) # Gaussian distribution
d = K.random_normal_variable(shape=(3, 4), mean=0, scale=1)

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

Backend functions

is_sparse

`keras.backend.is_sparse(tensor)`

Returns whether a tensor is a sparse tensor.

Arguments

- **tensor**: A tensor instance.

Returns

A boolean.

Example

```
>>> 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
```

---

to_dense

`keras.backend.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

```
>>> 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
```

---

variable

```
keras.backend.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

```
>>> 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.]])
```

---

constant

```
keras.backend.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.

---

is_keras_tensor

`keras.backend.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

```
>>> 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))
>>> K.is_keras_tensor(k_var) # A variable indirectly created outside of keras is not a Keras
False
>>> keras_var = K.variable(np_var)
>>> K.is_keras_tensor(keras_var)  # A variable created with the keras backend is not a Keras
False
>>> keras_placeholder = K.placeholder(shape=(2, 4, 5))
>>> K.is_keras_tensor(keras_placeholder)  # A placeholder is not a Keras tensor.
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)
>>> K.is_keras_tensor(keras_layer_output) # Any Keras layer output is a Keras tensor.
True
```

---

is_tensor

`keras.backend.is_tensor(x)`

---

placeholder

```
keras.backend.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

```
>>> 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>
```

---

is_placeholder

```
keras.backend.is_placeholder(x)
```

Returns whether `x` is a placeholder.

Arguments

- **x**: A candidate placeholder.

Returns

Boolean.

---

shape

```
keras.backend.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

```
### 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)
```

---

int_shape

`keras.backend.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

```
>>> 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)
```

---

ndim

`keras.backend.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

```
>>> 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
```

---

dtype

`keras.backend.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

```
>>> 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'
```

---

eval

`keras.backend.eval(x)`

Evaluates the value of a variable.

Arguments

- **x**: A variable.

Returns

A Numpy array.

Examples

```
>>> 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)
```

---

zeros

`keras.backend.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

```
>>> 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)
```

---

ones

`keras.backend.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

```
>>> 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)
```

---

eye

`keras.backend.eye(size, dtype=None, name=None)`

Instantiate an identity matrix and returns it.

Arguments

- **size**: Integer, number of rows/columns.
- **dtype**: String, data type of returned Keras variable.
- **name**: String, name of returned Keras variable.

Returns

A Keras variable, an identity matrix.

Example

```
>>> from keras import backend as K
>>> kvar = K.eye(3)
>>> K.eval(kvar)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]], dtype=float32)
```

---

zeros_like

```
keras.backend.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

```
>>> 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)
```

---

ones_like

```
keras.backend.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

```
>>> 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)
```

---

identity

```
keras.backend.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.

---

random_uniform_variable

```
keras.backend.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

```
### 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)
```

---

random_normal_variable

```
keras.backend.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

```
### 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)
```

---

count_params

```
keras.backend.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

```
>>> kvar = K.zeros((2,3))
>>> K.count_params(kvar)
6
>>> K.eval(kvar)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]], dtype=float32)
```

---

cast

`keras.backend.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

```
>>> 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>
```

---

update

`keras.backend.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.

---

update__add

`keras.backend.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.

---

update__sub

`keras.backend.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.

---

moving_average_update

`keras.backend.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.

---

dot

`keras.backend.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

```
### 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>
```

```
### 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>
```

```
### 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)
```

---

batch_dot

`keras.backend.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 batch, 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**: list of (or single) int with target dimensions. The lengths of `axes[0]` and `axes[1]` should be the same.

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 `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)`

```
>>> 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)
```

---

transpose

`keras.backend.transpose(x)`

Transposes a tensor and returns it.

Arguments

- **x**: Tensor or variable.

Returns

A tensor.

Examples

```
>>> 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)
```

```
>>> 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>
```

---

gather

`keras.backend.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`.

---

max

```
keras.backend.max(x, axis=None, keepdims=False)
```

Maximum value in a tensor.

Arguments

- **x**: A tensor or variable.
- **axis**: An integer, the axis to find maximum values.
- **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`.

---

min

```
keras.backend.min(x, axis=None, keepdims=False)
```

Minimum value in a tensor.

Arguments

- **x**: A tensor or variable.
- **axis**: An integer, the axis to find minimum values.
- **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`.

---

sum

```
keras.backend.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, the axis to sum over.
- **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`.

---

prod

```
keras.backend.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, the axis to compute the product.
- **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`.

---

cumsum

```
keras.backend.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`.

cumprod

```
keras.backend.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`.

---

var

```
keras.backend.var(x, axis=None, keepdims=False)
```

Variance of a tensor, alongside the specified axis.

Arguments

- **x**: A tensor or variable.
- **axis**: An integer, the axis to compute the variance.
- **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`.

---

std

```
keras.backend.std(x, axis=None, keepdims=False)
```

Standard deviation of a tensor, alongside the specified axis.

Arguments

- **x**: A tensor or variable.
- **axis**: An integer, the axis to compute the standard deviation.
- **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`.

---

mean

```
keras.backend.mean(x, axis=None, keepdims=False)
```

Mean of a tensor, alongside the specified axis.

Arguments

- **x**: A tensor or variable.
- **axis**: A list of integer. Axes to compute the mean.
- **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`.

---

any

```
keras.backend.any(x, axis=None, keepdims=False)
```

Bitwise reduction (logical OR).

Arguments

- **x**: Tensor or variable.
- **axis**: axis along which to perform the reduction.
- **keepdims**: whether the drop or broadcast the reduction axes.

Returns

A uint8 tensor (0s and 1s).

---

all

```
keras.backend.all(x, axis=None, keepdims=False)
```

Bitwise reduction (logical AND).

Arguments

- **x**: Tensor or variable.

- **axis**: axis along which to perform the reduction.
- **keepdims**: whether the drop or broadcast the reduction axes.

Returns

A uint8 tensor (0s and 1s).

---

argmax

`keras.backend.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.

---

argmin

`keras.backend.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.

---

square

`keras.backend.square(x)`

Element-wise square.

Arguments

- **x**: Tensor or variable.

Returns

A tensor.

---

abs

`keras.backend.abs(x)`

Element-wise absolute value.

Arguments

- **x**: Tensor or variable.

Returns

A tensor.

---

sqrt

`keras.backend.sqrt(x)`

Element-wise square root.

Arguments

- **x**: Tensor or variable.

Returns

A tensor.

---

exp

`keras.backend.exp(x)`

Element-wise exponential.

Arguments

- **x**: Tensor or variable.

Returns

A tensor.

---

log

`keras.backend.log(x)`

Element-wise log.

Arguments

- **x**: Tensor or variable.

Returns

A tensor.

---

logsumexp

`keras.backend.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**: An integer, the axis to reduce over.
- **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.

---

round

`keras.backend.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.

---

sign

`keras.backend.sign(x)`

Element-wise sign.

Arguments

- **x**: Tensor or variable.

Returns

A tensor.

---

pow

`keras.backend.pow(x, a)`

Element-wise exponentiation.

Arguments

- **x**: Tensor or variable.
- **a**: Python integer.

Returns

A tensor.

---

clip

`keras.backend.clip(x, min_value, max_value)`

Element-wise value clipping.

Arguments

- **x**: Tensor or variable.
- **min_value**: Python float or integer.
- **max_value**: Python float or integer.

Returns

A tensor.

---

equal

`keras.backend.equal(x, y)`

Element-wise equality between two tensors.

Arguments

- **x**: Tensor or variable.
- **y**: Tensor or variable.

Returns

A bool tensor.

---

not_equal

`keras.backend.not_equal(x, y)`

Element-wise inequality between two tensors.

Arguments

- **x**: Tensor or variable.
- **y**: Tensor or variable.

Returns

A bool tensor.

---

greater

`keras.backend.greater(x, y)`

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

Arguments

- **x**: Tensor or variable.
- **y**: Tensor or variable.

Returns

A bool tensor.

---

greater_equal

```
keras.backend.greater_equal(x, y)
```

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

Arguments

- **x**: Tensor or variable.
- **y**: Tensor or variable.

Returns

A bool tensor.

---

less

```
keras.backend.less(x, y)
```

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

Arguments

- **x**: Tensor or variable.
- **y**: Tensor or variable.

Returns

A bool tensor.

---

less_equal

```
keras.backend.less_equal(x, y)
```

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

Arguments

- **x**: Tensor or variable.
- **y**: Tensor or variable.

Returns

A bool tensor.

---

maximum

`keras.backend.maximum(x, y)`

Element-wise maximum of two tensors.

Arguments

- **x**: Tensor or variable.
- **y**: Tensor or variable.

Returns

A tensor.

---

minimum

`keras.backend.minimum(x, y)`

Element-wise minimum of two tensors.

Arguments

- **x**: Tensor or variable.
- **y**: Tensor or variable.

Returns

A tensor.

---

sin

`keras.backend.sin(x)`

Computes sin of x element-wise.

Arguments

- **x**: Tensor or variable.

Returns

A tensor.

---

cos

```
keras.backend.cos(x)
```

Computes cos of x element-wise.

Arguments

- **x**: Tensor or variable.

Returns

A tensor.

---

normalize_batch_in_training

```
keras.backend.normalize_batch_in_training(x, gamma, beta, reduction_axes, epsilon=0.001)
```

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)`.

---

batch_normalization

```
keras.backend.batch_normalization(x, mean, var, beta, gamma, epsilon=0.001)
```

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.
- **epsilon**: Fuzz factor.

Returns

A tensor.

---

concatenate

```
keras.backend.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.

---

reshape

```
keras.backend.reshape(x, shape)
```

Reshapes a tensor to the specified shape.

Arguments

- **x**: Tensor or variable.
- **shape**: Target shape tuple.

Returns

A tensor.

---

permute_dimensions

```
keras.backend.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.

resize_images

`keras.backend.resize_images(x, height_factor, width_factor, data_format)`

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"`.

Returns

A tensor.

Raises

- **ValueError**: if `data_format` is neither `"channels_last"` or `"channels_first"`.

---

resize_volumes

`keras.backend.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"`.

---

repeat_elements

```
keras.backend.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.

---

repeat

```
keras.backend.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.

---

arange

```
keras.backend.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.

---

tile

`keras.backend.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.

---

flatten

`keras.backend.flatten(x)`

Flatten a tensor.

Arguments

- **x**: A tensor or variable.

Returns

A tensor, reshaped into 1-D

---

batch_flatten

`keras.backend.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.

---

expand_dims

```
keras.backend.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.

---

squeeze

```
keras.backend.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.

---

temporal_padding

```
keras.backend.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.

---

spatial_2d_padding

```
keras.backend.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"`.

---

spatial_3d_padding

```
keras.backend.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"`.

---

stack

`keras.backend.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.

---

one_hot

`keras.backend.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)`

---

reverse

`keras.backend.reverse(x, axes)`

Reverse a tensor along the specified axes.

Arguments

- **x**: Tensor to reverse.
- **axes**: Integer or iterable of integers. Axes to reverse.

Returns

A tensor.

---

get_value

`keras.backend.get_value(x)`

Returns the value of a variable.

Arguments

- **x**: input variable.

Returns

A Numpy array.

---

batch_get_value

`keras.backend.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.

---

set_value

`keras.backend.set_value(x, value)`

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

Arguments

- **x**: Tensor to set to a new value.
- **value**: Value to set the tensor to, as a Numpy array (of the same shape).

---

batch_set_value

```
keras.backend.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

```
keras.backend.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

```
>>> 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.

---

function

```
keras.backend.function(inputs, outputs, updates=None)
```

Instantiates a Keras function.

Arguments

- **inputs**: List of placeholder tensors.
- **outputs**: List of output tensors.
- **updates**: List of update ops.
- ___**kwargs___: Passed to `tf.Session.run`.

Returns

Output values as Numpy arrays.

Raises

- **ValueError**: if invalid kwargs are passed in.

---

gradients

```
keras.backend.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.

---

stop_gradient

```
keras.backend.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.

---

rnn

```
keras.backend.rnn(step_function, inputs, initial_states, go_backwards=False, mask=None, cons
```

Iterates over the time dimension of a tensor.

Arguments

- **step_function**: RNN 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, output_dim)` (no time dimension).
- **new_states**: list of tensors, same length and shapes as 'states'. The first state in the list must be the output tensor at the previous timestep.
- **inputs**: tensor of temporal data of shape `(samples, time, ...)` (at least 3D).
- **initial_states**: tensor with shape (samples, output_dim) (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, 1)`, 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**: 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, ...)`.

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).

---

switch

`keras.backend.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.

---

in_train_phase

```
keras.backend.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()`.

---

in_test_phase

```
keras.backend.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`.

---

relu

```
keras.backend.relu(x, alpha=0.0, max_value=None)
```

Rectified linear unit.

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

Arguments

- **x**: A tensor or variable.
- **alpha**: A scalar, slope of negative section (default=`0.`).
- **max_value**: Saturation threshold.

Returns

A tensor.

---

elu

```
keras.backend.elu(x, alpha=1.0)
```

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.

---

softmax

```
keras.backend.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.

softplus

```
keras.backend.softplus(x)
```

Softplus of a tensor.

Arguments

- **x**: A tensor or variable.

Returns

A tensor.

---

softsign

```
keras.backend.softsign(x)
```

Softsign of a tensor.

Arguments

- **x**: A tensor or variable.

Returns

A tensor.

---

categorical_crossentropy

```
keras.backend.categorical_crossentropy(target, output, from_logits=False)
```

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.

Returns

Output tensor.

---

sparse_categorical_crossentropy

```
keras.backend.sparse_categorical_crossentropy(target, output, from_logits=False)
```

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.

Returns

Output tensor.

---

binary_crossentropy

```
keras.backend.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.

---

sigmoid

```
keras.backend.sigmoid(x)
```

Element-wise sigmoid.

Arguments

- **x**: A tensor or variable.

Returns

A tensor.

---

hard_sigmoid

```
keras.backend.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.

---

tanh

```
keras.backend.tanh(x)
```

Element-wise tanh.

Arguments

- **x**: A tensor or variable.

Returns

A tensor.

---

dropout

```
keras.backend.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.

---

l2_normalize

```
keras.backend.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.

---

in_top_k

```
keras.backend.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]`.

---

conv1d

```
keras.backend.conv1d(x, kernel, strides=1, padding='valid', data_format=None, dilation_rate=
```

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` or `channels_first`.

---

conv2d

```
keras.backend.conv2d(x, kernel, strides=(1, 1), padding='valid', data_format=None, dilation_
```

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` or `channels_first`.

---

conv2d_transpose

```
keras.backend.conv2d_transpose(x, kernel, output_shape, strides=(1, 1), padding='valid', dat
```

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.

Returns

A tensor, result of transposed 2D convolution.

Raises

- **ValueError**: if `data_format` is neither `channels_last` or `channels_first`.

---

separable_conv1d

```
keras.backend.separable_conv1d(x, depthwise_kernel, pointwise_kernel, strides=1, padding='va
```

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` or `channels_first`.

---

separable_conv2d

```
keras.backend.separable_conv2d(x, depthwise_kernel, pointwise_kernel, strides=(1, 1), paddin
```

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` or `channels_first`.

---

depthwise_conv2d

```
keras.backend.depthwise_conv2d(x, depthwise_kernel, strides=(1, 1), padding='valid', data_fo
```

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` or `channels_first`.

---

conv3d

```
keras.backend.conv3d(x, kernel, strides=(1, 1, 1), padding='valid', data_format=None, dilati
```

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` or `channels_first`.

---

conv3d_transpose

```
keras.backend.conv3d_transpose(x, kernel, output_shape, strides=(1, 1, 1), padding='valid',
```

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` or `channels_first`.

---

pool2d

```
keras.backend.pool2d(x, pool_size, strides=(1, 1), padding='valid', data_format=None, pool_r
```

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"`.

---

pool3d

```
keras.backend.pool3d(x, pool_size, strides=(1, 1, 1), padding='valid', data_format=None, poo
```

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"`.

---

bias_add

```
keras.backend.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

---

random_normal

```
keras.backend.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.

---

random_uniform

```
keras.backend.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.

---

random_binomial

```
keras.backend.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.

---

truncated_normal

```
keras.backend.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.

---

ctc_label_dense_to_sparse

```
keras.backend.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.

---

ctc__batch__cost

```
keras.backend.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.

---

ctc__decode

```
keras.backend.ctc_decode(y_pred, input_length, greedy=True, beam_width=100, top_paths=1)
```

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.

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.

---

map_fn

```
keras.backend.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`.

---

foldl

```
keras.backend.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`.

---

foldr

```
keras.backend.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`.

---

local_conv1d

```
keras.backend.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` or `channels_first`.

---

local_conv2d

```
keras.backend.local_conv2d(inputs, kernel, kernel_size, strides, output_shape, data_format=
```

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`.

---

manual_variable_initialization

`keras.backend.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 (e.g. via `tf.initialize_all_variables()`).

Arguments

- **value**: Python boolean.

---

learning_phase

`keras.backend.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).

---

set_learning_phase

`keras.backend.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`.

---

floatx

`keras.backend.floatx()`

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

Returns

String, the current default float type.

Example

```
>>> keras.backend.floatx()
'float32'
```

---

set_floatx

`keras.backend.set_floatx(floatx)`

Sets the default float type.

Arguments

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

Example

```
>>> from keras import backend as K
>>> K.floatx()
'float32'
>>> K.set_floatx('float16')
>>> K.floatx()
'float16'
```

---

cast_to_floatx

`keras.backend.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

```
>>> 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')
```

---

image_data_format

`keras.backend.image_data_format()`

Returns the default image data format convention ('channels_first' or 'channels_last').

Returns

A string, either `'channels_first'` or `'channels_last'`

Example

```
>>> keras.backend.image_data_format()
'channels_first'
```

---

set_image_data_format

`keras.backend.set_image_data_format(data_format)`

Sets the value of the data format convention.

Arguments

- **data_format**: string. `'channels_first'` or `'channels_last'`.

Example

```
>>> 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'
```

---

get_uid

`keras.backend.get_uid(prefix='')`

Get the uid for the default graph.

Arguments

- **prefix**: An optional prefix of the graph.

Returns

A unique identifier for the graph.

---

reset_uids

`keras.backend.reset_uids()`

Resets graph identifiers.

---

clear_session

`keras.backend.clear_session()`

Destroys the current TF graph and creates a new one.

Useful to avoid clutter from old models / layers.

---

backend

`backend.backend()`

Publicly accessible method for determining the current backend.

Returns

String, the name of the backend Keras is currently using.

Example

```
>>> keras.backend.backend()
'tensorflow'
```

---

epsilon

`keras.backend.epsilon()`

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

Returns

A float.

Example

```
>>> keras.backend.epsilon()
1e-07
```

---

set_epsilon

`keras.backend.set_epsilon(e)`

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

Arguments

- **e**: float. New value of epsilon.

Example

```
>>> from keras import backend as K
>>> K.epsilon()
1e-07
>>> K.set_epsilon(1e-05)
>>> K.epsilon()
1e-05
```

Initializers

Usage of initializers

Initializations define the way to set the initial random weights of Keras layers.

The keyword arguments used for passing initializers to layers will depend on the layer. Usually it is simply `kernel_initializer` and `bias_initializer`:

```
model.add(Dense(64,
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))
```

Available initializers

The following built-in initializers are available as part of the `keras.initializers` module:

[source] ##### TruncatedNormal

```
keras.initializers.TruncatedNormal(mean=0.0, stddev=0.05, seed=None)
```

Initializer that generates a truncated normal distribution.

These values are similar to values from a `RandomNormal` except that values more than two standard deviations from the mean are discarded and re-drawn. This is the recommended initializer for neural network weights and filters.

Arguments

- **mean**: a python scalar or a scalar tensor. Mean of the random values to generate.
- **stddev**: a python scalar or a scalar tensor. Standard deviation of the random values to generate.
- **seed**: A Python integer. Used to seed the random generator.

[source] ##### Ones

```
keras.initializers.Ones()
```

Initializer that generates tensors initialized to 1.

[source] ##### Initializer

```
keras.initializers.Initializer()
```

Initializer base class: all initializers inherit from this class.

[source] ##### RandomNormal

```
keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed=None)
```

Initializer that generates tensors with a normal distribution.

Arguments

- **mean**: a python scalar or a scalar tensor. Mean of the random values to generate.
- **stddev**: a python scalar or a scalar tensor. Standard deviation of the random values to generate.
- **seed**: A Python integer. Used to seed the random generator.

[source] ##### RandomUniform

```
keras.initializers.RandomUniform(minval=-0.05, maxval=0.05, seed=None)
```

Initializer that generates tensors with a uniform distribution.

Arguments

- **minval**: A python scalar or a scalar tensor. Lower bound of the range of random values to generate.
- **maxval**: A python scalar or a scalar tensor. Upper bound of the range of random values to generate. Defaults to 1 for float types.
- **seed**: A Python integer. Used to seed the random generator.

[source] ##### VarianceScaling

```
keras.initializers.VarianceScaling(scale=1.0, mode='fan_in', distribution='normal', seed=No
```

Initializer capable of adapting its scale to the shape of weights.

With `distribution="normal"`, samples are drawn from a truncated normal distribution centered on zero, with `stddev = sqrt(scale / n)` where n is:

- number of input units in the weight tensor, if mode = "fan_in"
- number of output units, if mode = "fan_out"
- average of the numbers of input and output units, if mode = "fan_avg"

With `distribution="uniform"`, samples are drawn from a uniform distribution within [-limit, limit], with `limit = sqrt(3 * scale / n)`.

Arguments

- **scale**: Scaling factor (positive float).
- **mode**: One of "fan_in", "fan_out", "fan_avg".
- **distribution**: Random distribution to use. One of "normal", "uniform".
- **seed**: A Python integer. Used to seed the random generator.

Raises

- **ValueError**: In case of an invalid value for the "scale", mode" or "distribution" arguments.

[source] ##### Orthogonal

```
keras.initializers.Orthogonal(gain=1.0, seed=None)
```

Initializer that generates a random orthogonal matrix.

Arguments

- **gain**: Multiplicative factor to apply to the orthogonal matrix.
- **seed**: A Python integer. Used to seed the random generator.

References

Saxe et al., http://arxiv.org/abs/1312.6120

[source] ##### Identity

```
keras.initializers.Identity(gain=1.0)
```

Initializer that generates the identity matrix.

Only use for square 2D matrices.

Arguments

- **gain**: Multiplicative factor to apply to the identity matrix.

[source] ##### Constant

```
keras.initializers.Constant(value=0)
```

Initializer that generates tensors initialized to a constant value.

Arguments

- **value**: float; the value of the generator tensors.

[source] ##### Zeros

```
keras.initializers.Zeros()
```

Initializer that generates tensors initialized to 0.

---

glorot_normal

```
glorot_normal(seed=None)
```

Glorot normal initializer, also called Xavier normal initializer.

It draws samples from a truncated normal distribution centered on 0 with `stddev = sqrt(2 / (fan_in + fan_out))` where `fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units in the weight tensor.

Arguments

- **seed**: A Python integer. Used to seed the random generator.

Returns

An initializer.

References

Glorot & Bengio, AISTATS 2010 - **http**://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf

---

glorot_uniform

```
glorot_uniform(seed=None)
```

Glorot uniform initializer, also called Xavier uniform initializer.

It draws samples from a uniform distribution within [-limit, limit] where `limit` is `sqrt(6 / (fan_in + fan_out))` where `fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units in the weight tensor.

Arguments

- **seed**: A Python integer. Used to seed the random generator.

Returns

An initializer.

References

Glorot & Bengio, AISTATS 2010 - **http**://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf

---

he_normal

```
he_normal(seed=None)
```

He normal initializer.

It draws samples from a truncated normal distribution centered on 0 with `stddev = sqrt(2 / fan_in)` where `fan_in` is the number of input units in the weight tensor.

Arguments

- **seed**: A Python integer. Used to seed the random generator.

Returns

An initializer.

References

He et al., http://arxiv.org/abs/1502.01852

---

lecun_normal

```
lecun_normal(seed=None)
```

LeCun normal initializer.

It draws samples from a truncated normal distribution centered on 0 with `stddev = sqrt(1 / fan_in)` where `fan_in` is the number of input units in the weight tensor.

Arguments

- **seed**: A Python integer. Used to seed the random generator.

Returns

An initializer.

References

- Self-Normalizing Neural Networks
- Efficient Backprop

---

he_uniform

```
he_uniform(seed=None)
```

He uniform variance scaling initializer.

It draws samples from a uniform distribution within [-limit, limit] where `limit` is `sqrt(6 / fan_in)` where `fan_in` is the number of input units in the weight tensor.

Arguments

- **seed**: A Python integer. Used to seed the random generator.

Returns

An initializer.

References

He et al., http://arxiv.org/abs/1502.01852

---

lecun_uniform

```
lecun_uniform(seed=None)
```

LeCun uniform initializer.

It draws samples from a uniform distribution within [-limit, limit] where `limit` is `sqrt(3 / fan_in)` where `fan_in` is the number of input units in the weight tensor.

Arguments

- **seed**: A Python integer. Used to seed the random generator.

Returns

An initializer.

References

LeCun 98, Efficient Backprop, - **http**://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

An initializer may be passed as a string (must match one of the available initializers above), or as a callable:

```
from keras import initializers

model.add(Dense(64, kernel_initializer=initializers.random_normal(stddev=0.01)))

### also works; will use the default parameters.
model.add(Dense(64, kernel_initializer='random_normal'))
```

Using custom initializers

If passing a custom callable, then it must take the argument `shape` (shape of the variable to initialize) and `dtype` (dtype of generated values):

```
from keras import backend as K

def my_init(shape, dtype=None):
    return K.random_normal(shape, dtype=dtype)

model.add(Dense(64, kernel_initializer=my_init))
```

Regularizers

Usage of regularizers

Regularizers allow to apply penalties on layer parameters or layer activity during optimization. These penalties are incorporated in the loss function that the network optimizes.

The penalties are applied on a per-layer basis. The exact API will depend on the layer, but the layers `Dense`, `Conv1D`, `Conv2D` and `Conv3D` have a unified API.

These layers expose 3 keyword arguments:

- `kernel_regularizer`: instance of `keras.regularizers.Regularizer`
- `bias_regularizer`: instance of `keras.regularizers.Regularizer`
- `activity_regularizer`: instance of `keras.regularizers.Regularizer`

Example

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01),
                activity_regularizer=regularizers.l1(0.01)))
```

Available penalties

```
keras.regularizers.l1(0.)
keras.regularizers.l2(0.)
keras.regularizers.l1_l2(0.)
```

Developing new regularizers

Any function that takes in a weight matrix and returns a loss contribution tensor can be used as a regularizer, e.g.:

```
from keras import backend as K

def l1_reg(weight_matrix):
    return 0.01 * K.sum(K.abs(weight_matrix))
```

```
model.add(Dense(64, input_dim=64,
                kernel_regularizer=l1_reg))
```

Alternatively, you can write your regularizers in an object-oriented way; see the keras/regularizers.py module for examples.

Constraints

Usage of constraints

Functions from the `constraints` module allow setting constraints (eg. non-negativity) on network parameters during optimization.

The penalties are applied on a per-layer basis. The exact API will depend on the layer, but the layers `Dense`, `Conv1D`, `Conv2D` and `Conv3D` have a unified API.

These layers expose 2 keyword arguments:

- `kernel_constraint` for the main weights matrix
- `bias_constraint` for the bias.

```
from keras.constraints import max_norm
model.add(Dense(64, kernel_constraint=max_norm(2.)))
```

Available constraints

- **max_norm(max_value=2, axis=0)**: maximum-norm constraint
- **non_neg()**: non-negativity constraint
- **unit_norm(axis=0)**: unit-norm constraint
- **min_max_norm(min_value=0.0, max_value=1.0, rate=1.0, axis=0)**: minimum/maximum-norm constraint

Visualization

Model visualization

The `keras.utils.vis_utils` module provides utility functions to plot a Keras model (using `graphviz`).

This will plot a graph of the model and save it to a file:

```
from keras.utils import plot_model
plot_model(model, to_file='model.png')
```

`plot_model` takes two optional arguments:

- `show_shapes` (defaults to False) controls whether output shapes are shown in the graph.
- `show_layer_names` (defaults to True) controls whether layer names are shown in the graph.

You can also directly obtain the `pydot.Graph` object and render it yourself, for example to show it in an ipython notebook :

```
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot

SVG(model_to_dot(model).create(prog='dot', format='svg'))
```

Scikit-learn API

Wrappers for the Scikit-Learn API

You can use `Sequential` Keras models (single-input only) as part of your Scikit-Learn workflow via the wrappers found at `keras.wrappers.scikit_learn.py`.

There are two wrappers available:

`keras.wrappers.scikit_learn.KerasClassifier(build_fn=None, **sk_params)`, which implements the Scikit-Learn classifier interface,

`keras.wrappers.scikit_learn.KerasRegressor(build_fn=None, **sk_params)`, which implements the Scikit-Learn regressor interface.

Arguments

- **build__fn**: callable function or class instance
- **sk__params**: model parameters & fitting parameters

`build_fn` should construct, compile and return a Keras model, which will then be used to fit/predict. One of the following three values could be passed to `build_fn`:

1. A function
2. An instance of a class that implements the `__call__` method
3. None. This means you implement a class that inherits from either `KerasClassifier` or `KerasRegressor`. The `__call__` method of the present class will then be treated as the default `build_fn`.

`sk_params` takes both model parameters and fitting parameters. Legal model parameters are the arguments of `build_fn`. Note that like all other estimators in scikit-learn, `build_fn` should provide default values for its arguments, so that you could create the estimator without passing any values to `sk_params`.

`sk_params` could also accept parameters for calling `fit`, `predict`, `predict_proba`, and `score` methods (e.g., `epochs`, `batch_size`). fitting (predicting) parameters are selected in the following order:

1. Values passed to the dictionary arguments of `fit`, `predict`, `predict_proba`, and `score` methods
2. Values passed to `sk_params`
3. The default values of the `keras.models.Sequential` `fit`, `predict`, `predict_proba` and `score` methods

When using scikit-learn's `grid_search` API, legal tunable parameters are those you could pass to `sk_params`, including fitting parameters. In other words, you could use `grid_search` to search for the best `batch_size` or `epochs` as well as the model parameters.

Utils

[source] ##### CustomObjectScope

`keras.utils.CustomObjectScope()`

Provides a scope that changes to `_GLOBAL_CUSTOM_OBJECTS` cannot escape.

Code within a `with` statement will be able to access custom objects by name. Changes to global custom objects persist within the enclosing `with` statement. At end of the `with` statement, global custom objects are reverted to state at beginning of the `with` statement.

Example

Consider a custom object `MyObject` (e.g. a class):

```
with CustomObjectScope({'MyObject':MyObject}):
    layer = Dense(..., kernel_regularizer='MyObject')
    # save, load, etc. will recognize custom object by name
```

[source] ##### HDF5Matrix

`keras.utils.HDF5Matrix(datapath, dataset, start=0, end=None, normalizer=None)`

Representation of HDF5 dataset to be used instead of a Numpy array.

Example

```
x_data = HDF5Matrix('input/file.hdf5', 'data')
model.predict(x_data)
```

Providing `start` and `end` allows use of a slice of the dataset.

Optionally, a normalizer function (or lambda) can be given. This will be called on every slice of data retrieved.

Arguments

- **datapath**: string, path to a HDF5 file
- **dataset**: string, name of the HDF5 dataset in the file specified in datapath
- **start**: int, start of desired slice of the specified dataset
- **end**: int, end of desired slice of the specified dataset
- **normalizer**: function to be called on data when retrieved

Returns

An array-like HDF5 dataset.

[source] ##### Sequence

`keras.utils.Sequence()`

Base object for fitting to a sequence of data, such as a dataset.

Every `Sequence` must implement the `__getitem__` and the `__len__` methods. If you want to modify your dataset between epochs you may implement `on_epoch_end`. The method `__getitem__` should return a complete batch.

Notes

`Sequence` are a safer way to do multiprocessing. This structure guarantees that the network will only train once on each sample per epoch which is not the case with generators.

Examples

```
from skimage.io import imread
from skimage.transform import resize
import numpy as np

### Here, `x_set` is list of path to the images
### and `y_set` are the associated classes.

class CIFAR10Sequence(Sequence):

    def __init__(self, x_set, y_set, batch_size):
        self.x, self.y = x_set, y_set
        self.batch_size = batch_size

    def __len__(self):
        return int(np.ceil(len(self.x) / float(self.batch_size)))

    def __getitem__(self, idx):
        batch_x = self.x[idx * self.batch_size:(idx + 1) * self.batch_size]
        batch_y = self.y[idx * self.batch_size:(idx + 1) * self.batch_size]

        return np.array([
            resize(imread(file_name), (200, 200))
               for file_name in batch_x]), np.array(batch_y)
```

---

to_categorical

```
keras.utils.to_categorical(y, num_classes=None)
```

Converts a class vector (integers) to binary class matrix.

E.g. for use with categorical_crossentropy.

Arguments

- **y**: class vector to be converted into a matrix (integers from 0 to num_classes).
- **num_classes**: total number of classes.

Returns

A binary matrix representation of the input.

---

normalize

```
keras.utils.normalize(x, axis=-1, order=2)
```

Normalizes a Numpy array.

Arguments

- **x**: Numpy array to normalize.
- **axis**: axis along which to normalize.
- **order**: Normalization order (e.g. 2 for L2 norm).

Returns

A normalized copy of the array.

---

get_file

```
keras.utils.get_file(fname, origin, untar=False, md5_hash=None, file_hash=None, cache_subdi
```

Downloads a file from a URL if it not already in the cache.

By default the file at the url `origin` is downloaded to the cache_dir `~/.keras`, placed in the cache_subdir `datasets`, and given the filename `fname`. The final location of a file `example.txt` would therefore be `~/.keras/datasets/example.txt`.

Files in tar, tar.gz, tar.bz, and zip formats can also be extracted. Passing a hash will verify the file after download. The command line programs `shasum` and `sha256sum` can compute the hash.

Arguments

- **fname**: Name of the file. If an absolute path `/path/to/file.txt` is specified the file will be saved at that location.
- **origin**: Original URL of the file.
- **untar**: Deprecated in favor of 'extract'. boolean, whether the file should be decompressed
- **md5_hash**: Deprecated in favor of 'file_hash'. md5 hash of the file for verification
- **file_hash**: The expected hash string of the file after download. The sha256 and md5 hash algorithms are both supported.
- **cache_subdir**: Subdirectory under the Keras cache dir where the file is saved. If an absolute path `/path/to/folder` is specified the file will be saved at that location.
- **hash_algorithm**: Select the hash algorithm to verify the file. options are 'md5', 'sha256', and 'auto'. The default 'auto' detects the hash algorithm in use.
- **extract**: True tries extracting the file as an Archive, like tar or zip.
- **archive_format**: Archive format to try for extracting the file. Options are 'auto', 'tar', 'zip', and None. 'tar' includes tar, tar.gz, and tar.bz files. The default 'auto' is ['tar', 'zip']. None or an empty list will return no matches found.
- **cache_dir**: Location to store cached files, when None it defaults to the Keras Directory.

Returns

Path to the downloaded file

---

print_summary

```
keras.utils.print_summary(model, line_length=None, positions=None, print_fn=None)
```

Prints a summary of a model.

Arguments

- **model**: Keras model instance.
- **line_length**: Total length of printed lines (e.g. set this to adapt the display to different terminal window sizes).
- **positions**: Relative or absolute positions of log elements in each line. If not provided, defaults to `[.33, .55, .67, 1.]`.
- **print_fn**: Print function to use. It will be called on each line of the summary. You can set it to a custom function in order to capture the string summary. It defaults to `print` (prints to stdout).

---

plot_model

```
keras.utils.plot_model(model, to_file='model.png', show_shapes=False, show_layer_names=True,
```

Converts a Keras model to dot format and save to a file.

Arguments

- **model**: A Keras model instance
- **to_file**: File name of the plot image.
- **show_shapes**: whether to display shape information.
- **show_layer_names**: whether to display layer names.
- **rankdir**: `rankdir` argument passed to PyDot, a string specifying the format of the plot: 'TB' creates a vertical plot; 'LR' creates a horizontal plot.

---

multi_gpu_model

```
keras.utils.multi_gpu_model(model, gpus=None, cpu_merge=True, cpu_relocation=False)
```

Replicates a model on different GPUs.

Specifically, this function implements single-machine multi-GPU data parallelism. It works in the following way:

- Divide the model's input(s) into multiple sub-batches.
- Apply a model copy on each sub-batch. Every model copy is executed on a dedicated GPU.
- Concatenate the results (on CPU) into one big batch.

E.g. if your `batch_size` is 64 and you use `gpus=2`, then we will divide the input into 2 sub-batches of 32 samples, process each sub-batch on one GPU, then return the full batch of 64 processed samples.

This induces quasi-linear speedup on up to 8 GPUs.

This function is only available with the TensorFlow backend for the time being.

Arguments

- **model**: A Keras model instance. To avoid OOM errors, this model could have been built on CPU, for instance (see usage example below).
- **gpus**: Integer >= 2 or list of integers, number of GPUs or list of GPU IDs on which to create model replicas.
- **cpu_merge**: A boolean value to identify whether to force merging model weights under the scope of the CPU or not.
- **cpu_relocation**: A boolean value to identify whether to create the model's weights under the scope of the CPU. If the model is not defined under any preceding device scope, you can still rescue it by activating this option.

Returns

A Keras `Model` instance which can be used just like the initial `model` argument, but which distributes its workload on multiple GPUs.

Example 1 - Training models with weights merge on CPU

```
import tensorflow as tf
from keras.applications import Xception
from keras.utils import multi_gpu_model
import numpy as np

num_samples = 1000
height = 224
width = 224
num_classes = 1000

### Instantiate the base model (or "template" model).
### We recommend doing this with under a CPU device scope,
### so that the model's weights are hosted on CPU memory.
### Otherwise they may end up hosted on a GPU, which would
### complicate weight sharing.
with tf.device('/cpu:0'):
    model = Xception(weights=None,
                     input_shape=(height, width, 3),
                     classes=num_classes)

### Replicates the model on 8 GPUs.
### This assumes that your machine has 8 available GPUs.
parallel_model = multi_gpu_model(model, gpus=8)
parallel_model.compile(loss='categorical_crossentropy',
                       optimizer='rmsprop')

### Generate dummy data.
x = np.random.random((num_samples, height, width, 3))
```

```
y = np.random.random((num_samples, num_classes))

### This `fit` call will be distributed on 8 GPUs.
### Since the batch size is 256, each GPU will process 32 samples.
parallel_model.fit(x, y, epochs=20, batch_size=256)

### Save model via the template model (which shares the same weights):
model.save('my_model.h5')
```

Example 2 - Training models with weights merge on CPU using cpu_relocation

```
..
### Not needed to change the device scope for model definition:
model = Xception(weights=None, ..)

try:
    model = multi_gpu_model(model, cpu_relocation=True)
    print("Training using multiple GPUs..")
except:
    print("Training using single GPU or CPU..")

model.compile(..)
..
```

Example 3 - Training models with weights merge on GPU (recommended for NV-link)

```
..
### Not needed to change the device scope for model definition:
model = Xception(weights=None, ..)

try:
    model = multi_gpu_model(model, cpu_merge=False)
    print("Training using multiple GPUs..")
except:
    print("Training using single GPU or CPU..")

model.compile(..)
..
```

On model saving

To save the multi-gpu model, use `.save(fname)` or `.save_weights(fname)` with the template model (the argument you passed to `multi_gpu_model`), rather than the model returned by `multi_gpu_model`.

Contributing

On Github Issues and Pull Requests

Found a bug? Have a new feature to suggest? Want to contribute changes to the codebase? Make sure to read this first.

Bug reporting

Your code doesn't work, and you have determined that the issue lies with Keras? Follow these steps to report a bug.

1. Your bug may already be fixed. Make sure to update to the current Keras master branch, as well as the latest Theano/TensorFlow/CNTK master branch. To easily update Theano: `pip install git+git://github.com/Theano/Theano.git --upgrade`
2. Search for similar issues. Make sure to delete `is:open` on the issue search to find solved tickets as well. It's possible somebody has encountered this bug already. Also remember to check out Keras' FAQ. Still having a problem? Open an issue on Github to let us know.
3. Make sure you provide us with useful information about your configuration: what OS are you using? What Keras backend are you using? Are you running on GPU? If so, what is your version of Cuda, of cuDNN? What is your GPU?
4. Provide us with a script to reproduce the issue. This script should be runnable as-is and should not require external data download (use randomly generated data if you need to run a model on some test data). We recommend that you use Github Gists to post your code. Any issue that cannot be reproduced is likely to be closed.
5. If possible, take a stab at fixing the bug yourself –if you can!

The more information you provide, the easier it is for us to validate that there is a bug and the faster we'll be able to take action. If you want your issue to be resolved quickly, following the steps above is crucial.

Requesting a Feature

You can also use Github issues to request features you would like to see in Keras, or changes in the Keras API.

1. Provide a clear and detailed explanation of the feature you want and why it's important to add. Keep in mind that we want features that will be useful to the majority of our users and not just a small subset. If you're

just targeting a minority of users, consider writing an add-on library for Keras. It is crucial for Keras to avoid bloating the API and codebase.

2. Provide code snippets demonstrating the API you have in mind and illustrating the use cases of your feature. Of course, you don't need to write any real code at this point!
3. After discussing the feature you may choose to attempt a Pull Request. If you're at all able, start writing some code. We always have more work to do than time to do it. If you can write some code then that will speed the process along.

Requests for Contributions

This is the board where we list current outstanding issues and features to be added. If you want to start contributing to Keras, this is the place to start.

Pull Requests

Where should I submit my pull request?

1. **Keras improvements and bugfixes** go to the Keras `master` branch.
2. **Experimental new features** such as layers and datasets go to keras-contrib. Unless it is a new feature listed in Requests for Contributions, in which case it belongs in core Keras. If you think your feature belongs in core Keras, you can submit a design doc to explain your feature and argue for it (see explanations below).

Please note that PRs that are primarily about **code style** (as opposed to fixing bugs, improving docs, or adding new functionality) will likely be rejected.

Here's a quick guide to submitting your improvements:

1. If your PR introduces a change in functionality, make sure you start by writing a design doc and sending it to the Keras mailing list to discuss whether the change should be made, and how to handle it. This will save you from having your PR closed down the road! Of course, if your PR is a simple bug fix, you don't need to do that. The process for writing and submitting design docs is as follow:
   - Start from this Google Doc template, and copy it to new Google doc.
   - Fill in the content. Note that you will need to insert code examples. To insert code, use a Google Doc extension such as CodePretty (there are several such extensions available).
   - Set sharing settings to "everyone with the link is allowed to comment"

    - Send the document to `keras-users@googlegroups.com` with a subject that starts with `[API DESIGN REVIEW]` (all caps) so that we notice it.
    - Wait for comments, and answer them as they come. Edit the proposal as necessary.
    - The proposal will finally be approved or rejected. Once approved, you can send out Pull Requests or ask others to write Pull Requests.

2. Write the code (or get others to write it). This is the hard part!

3. Make sure any new function or class you introduce has proper docstrings. Make sure any code you touch still has up-to-date docstrings and documentation. **Docstring style should be respected.** In particular, they should be formatted in MarkDown, and there should be sections for `Arguments`, `Returns`, `Raises` (if applicable). Look at other docstrings in the codebase for examples.

4. Write tests. Your code should have full unit test coverage. If you want to see your PR merged promptly, this is crucial.

5. Run our test suite locally. It's easy: from the Keras folder, simply run: `py.test tests/`.

    - You will need to install the test requirements as well: `pip install -e .[tests]`.

6. Make sure all tests are passing:

    - with the Theano backend, on Python 2.7 and Python 3.6. Make sure you have the development version of Theano.
    - with the TensorFlow backend, on Python 2.7 and Python 3.6. Make sure you have the development version of TensorFlow.
    - with the CNTK backend, on Python 2.7 and Python 3.6. Make sure you have the development version of CNTK.

7. We use PEP8 syntax conventions, but we aren't dogmatic when it comes to line length. Make sure your lines stay reasonably sized, though. To make your life easier, we recommend running a PEP8 linter:

    - Install PEP8 packages: `pip install pep8 pytest-pep8 autopep8`
    - Run a standalone PEP8 check: `py.test --pep8 -m pep8`
    - You can automatically fix some PEP8 error by running: `autopep8 -i --select <errors> <FILENAME>` for example: `autopep8 -i --select E128 tests/keras/backend/test_backends.py`

8. When committing, use appropriate, descriptive commit messages.

9. Update the documentation. If introducing new functionality, make sure you include code snippets demonstrating the usage of your new feature.

10. Submit your PR. If your changes have been approved in a previous discussion, and if you have complete (and passing) unit tests as well as proper

docstrings/documentation, your PR is likely to be merged promptly.

Adding new examples

Even if you don't contribute to the Keras source code, if you have an application of Keras that is concise and powerful, please consider adding it to our collection of examples. Existing examples show idiomatic Keras code: make sure to keep your own script in the same spirit.