**Keras documentation for image preprocessing**

**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**

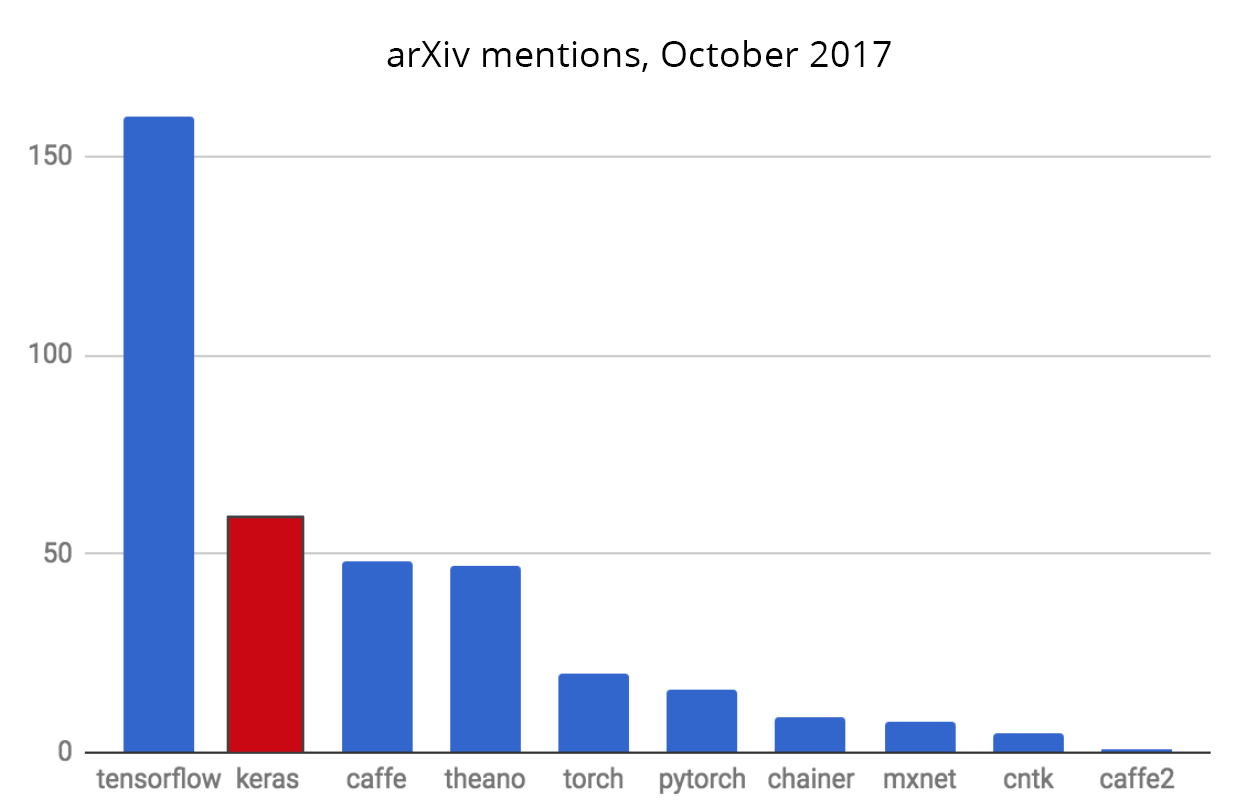
* Keras is an API designed for human beings, not machines. [Keras follows best practices for reducing cognitive load](https://blog.keras.io/user-experience-design-for-apis.html): it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides 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](https://www.quora.com/Why-has-Keras-been-so-successful-lately-at-Kaggle-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](https://arxiv.org/archive/cs):



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](https://developer.apple.com/documentation/coreml) (Keras support officially provided by Apple). Here's [a tutorial](https://www.pyimagesearch.com/2018/04/23/running-keras-models-on-ios-with-coreml/).
* On Android, via the TensorFlow Android runtime. Example: [Not Hotdog app](https://medium.com/@timanglade/how-hbos-silicon-valley-built-not-hotdog-with-mobile-tensorflow-keras-react-native-ef03260747f3).
* In the browser, via GPU-accelerated JavaScript runtimes such as [Keras.js](https://transcranial.github.io/keras-js/#/) and [WebDNN](https://mil-tokyo.github.io/webdnn/).
* On Google Cloud, via [TensorFlow-Serving](https://www.tensorflow.org/serving/).
* [In a Python webapp backend (such as a Flask app)](https://blog.keras.io/building-a-simple-keras-deep-learning-rest-api.html).
* On the JVM, via [DL4J model import provided by SkyMind](https://deeplearning4j.org/model-import-keras).
* 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](https://keras.io/backend/). 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:

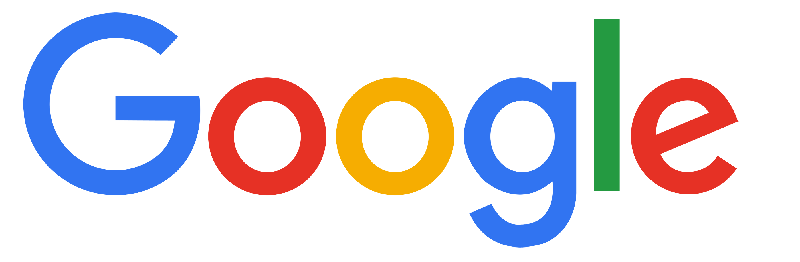
* [NVIDIA GPUs](https://developer.nvidia.com/deep-learning)
* [Google TPUs](https://cloud.google.com/tpu/), via the TensorFlow backend and Google Cloud
* OpenCL-enabled GPUs, such as those from AMD, via [the PlaidML Keras backend](https://github.com/plaidml/plaidml)

**Keras has strong multi-GPU support and distributed training support**

* Keras has [built-in support for multi-GPU data parallelism](https://keras.io/utils/#multi_gpu_model)
* [Horovod](https://github.com/uber/horovod), from Uber, has first-class support for Keras models
* Keras models [can be turned into TensorFlow Estimators](https://www.tensorflow.org/versions/master/api_docs/python/tf/keras/estimator/model_to_estimator) and trained on [clusters of GPUs on Google Cloud](https://cloud.google.com/solutions/running-distributed-tensorflow-on-compute-engine)
* Keras can be run on Spark via [Dist-Keras](https://github.com/cerndb/dist-keras) (from CERN) and [Elephas](https://github.com/maxpumperla/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).

**MODELS:**

**About Keras models**

There are two types of models available in Keras: [the Sequential model](https://keras.io/models/sequential) and [the Model class used with functional API](https://keras.io/models/model).

These models have a number of methods in common:

* model.summary(): prints a summary representation of your model. Shortcut for [utils.print\_summary](https://keras.io/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:

**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:

**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?](https://keras.io/getting-started/faq/#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](https://keras.io/getting-started/sequential-model-guide).

## Useful attributes of Model

* model.layers is a list of the layers added to the model.

## Sequential model methods

### compile

compile(self, optimizer, loss, metrics=**None**, sample\_weight\_mode=**None**, weighted\_metrics=**None**, target\_tensors=**None**)

Configures the model for training.

**Arguments**

* **optimizer**: String (name of optimizer) or optimizer object. See [optimizers](https://keras.io/optimizers).
* **loss**: String (name of objective function) or objective function. See [losses](https://keras.io/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'}.
* **sample\_weight\_mode**: If you need to do timestep-wise sample weighting (2D weights), set this to "temporal". Nonedefaults 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 a placeholder 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 tensor (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 should be a single tensor (for a single-output Sequential model).
* **\*\*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.

**Example**

model = Sequential()

model.add(Dense(32, input\_shape=(500,)))

model.add(Dense(10, activation='softmax'))

model.compile(optimizer='rmsprop',

loss='categorical\_crossentropy',

metrics=['accuracy'])

### fit

fit(self, x=**None**, y=**None**, batch\_size=**None**, epochs=1, verbose=1, callbacks=**None**, validation\_split=0.0, validation\_data=**None**, shuffle=**True**, class\_weight=**None**, sample\_weight=**None**, initial\_epoch=0, steps\_per\_epoch=**None**, validation\_steps=**None**)

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

**Arguments**

* **x**: Numpy array of training data. If the input layer in the model is named, you can also pass a dictionary mapping the input name to a Numpy array. 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 output layer in the model is named, you can also pass a dictionary mapping the output name to a Numpy array. 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, it 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**: 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](https://keras.io/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. This 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 specifysample\_weight\_mode="temporal" in compile().
* **initial\_epoch**: Epoch at which to start training (useful for resuming a previous training run).
* **steps\_per\_epoch**: 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**)

Computes the loss on some input data, batch by batch.

**Arguments**

* **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs). x can be None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
* **y**: labels, as a Numpy array. y can be None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
* **batch\_size**: Integer. If unspecified, it will default to 32.
* **verbose**: verbosity mode, 0 or 1.
* **sample\_weight**: sample weights, as a Numpy array.
* **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 no metrics) or list of scalars (if the model computes other metrics). The attribute model.metrics\_names will give you the display labels for the scalar outputs.

**Raises**

* **RuntimeError**: if the model was never compiled.

### predict

predict(self, x, batch\_size=**None**, verbose=0, steps=**None**)

Generates output predictions for the input samples.

The input samples are processed batch by batch.

**Arguments**

* **x**: the input data, as a Numpy array.
* **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**

A Numpy array of predictions.

### train\_on\_batch

train\_on\_batch(self, x, y, class\_weight=**None**, sample\_weight=**None**)

Single gradient update over one batch of samples.

**Arguments**

* **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
* **y**: labels, as a Numpy array.
* **class\_weight**: dictionary mapping classes to a weight value, used for scaling the loss function (during training only).
* **sample\_weight**: sample weights, as a Numpy array.

**Returns**

Scalar training loss (if the model has no metrics) or list of scalars (if the model computes other metrics). The attribute model.metrics\_names will give you the display labels for the scalar outputs.

**Raises**

* **RuntimeError**: if the model was never compiled.

### test\_on\_batch

test\_on\_batch(self, x, y, sample\_weight=**None**)

Evaluates the model over a single batch of samples.

**Arguments**

* **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
* **y**: labels, as a Numpy array.
* **sample\_weight**: sample weights, as a Numpy array.

**Returns**

Scalar test loss (if the model has no metrics) or list of scalars (if the model computes other metrics). The attribute model.metrics\_names will give you the display labels for the scalar outputs.

**Raises**

* **RuntimeError**: if the model was never compiled.

### predict\_on\_batch

predict\_on\_batch(self, x)

Returns predictions for a single batch of samples.

**Arguments**

* **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).

**Returns**

A Numpy array of predictions.

### fit\_generator

fit\_generator(self, generator, steps\_per\_epoch=**None**, epochs=1, verbose=1, callbacks=**None**, validation\_data=**None**, validation\_steps=**None**, class\_weight=**None**, max\_queue\_size=10, workers=1, use\_multiprocessing=**False**, shuffle=**True**, initial\_epoch=0)

Fits the model on data generated batch-by-batch by a Python generator.

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**: 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, total number of iterations on the data. Note that in conjunction with initial\_epoch, the parameter epochs is to be understood as "final epoch". The model is not trained for n steps given by epochs, but until the epoch 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](https://keras.io/callbacks).
* **validation\_data**: This can be either
* a generator for the validation data
* a tuple (inputs, targets)
* a tuple (inputs, targets, sample\_weights).
* **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, workerswill 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.

**Raises**

* **RuntimeError**: if the model was never compiled.
* **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*

x, y = process\_line(line)

**yield** (x, y)

model.fit\_generator(generate\_arrays\_from\_file('/my\_file.txt'),

steps\_per\_epoch=1000, epochs=10)

### evaluate\_generator

evaluate\_generator(self, generator, steps=**None**, max\_queue\_size=10, workers=1, use\_multiprocessing=**False**)

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)
* **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**: maximum number of processes to spin up
* **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.

**Returns**

Scalar test loss (if the model has no metrics) or list of scalars (if the model computes other metrics). The attribute model.metrics\_names will give you the display labels for the scalar outputs.

**Raises**

* **RuntimeError**: if the model was never compiled.

### predict\_generator

predict\_generator(self, generator, steps=**None**, max\_queue\_size=10, workers=1, use\_multiprocessing=**False**, verbose=0)

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.
* **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**: maximum number of processes to spin up
* **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**

A Numpy array of predictions.

### get\_layer

get\_layer(self, name=**None**, index=**None**)

Retrieve a layer that is part of the model.

Returns a layer based on either its name (unique) or its index in the graph. 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.

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](https://keras.io/getting-started/functional-api-guide/#the-concept-of-layer-node)), 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)

### Dense

keras.layers.Dense(units, activation=**None**, use\_bias=**True**, kernel\_initializer='glorot\_uniform', bias\_initializer='zeros', kernel\_regularizer=**None**, bias\_regularizer=**None**, activity\_regularizer=**None**, kernel\_constraint=**None**, bias\_constraint=**None**)

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](https://keras.io/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](https://keras.io/initializers/)).
* **bias\_initializer**: Initializer for the bias vector (see [initializers](https://keras.io/initializers/)).
* **kernel\_regularizer**: Regularizer function applied to the kernel weights matrix (see [regularizer](https://keras.io/regularizers/)).
* **bias\_regularizer**: Regularizer function applied to the bias vector (see [regularizer](https://keras.io/regularizers/)).
* **activity\_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see [regularizer](https://keras.io/regularizers/)).
* **kernel\_constraint**: Constraint function applied to the kernel weights matrix (see [constraints](https://keras.io/constraints/)).
* **bias\_constraint**: Constraint function applied to the bias vector (see [constraints](https://keras.io/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).

### Activation

keras.layers.Activation(activation)

Applies an activation function to an output.

**Arguments**

* **activation**: name of activation function to use (see: [activations](https://keras.io/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.

### 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](http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf)

### 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)*

### Input

keras.engine.topology.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)

### 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)*

### 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.

### 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).

### 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).

### 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.

### 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))

Convolution layers:

### Conv1D

keras.layers.Conv1D(filters, kernel\_size, strides=1, padding='valid', dilation\_rate=1, activation=**None**, use\_bias=**True**, kernel\_initializer='glorot\_uniform', bias\_initializer='zeros', kernel\_regularizer=**None**, bias\_regularizer=**None**, activity\_regularizer=**None**, kernel\_constraint=**None**, bias\_constraint=**None**)

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](https://arxiv.org/abs/1609.03499).
* **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](https://keras.io/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](https://keras.io/initializers/)).
* **bias\_initializer**: Initializer for the bias vector (see [initializers](https://keras.io/initializers/)).
* **kernel\_regularizer**: Regularizer function applied to the kernel weights matrix (see [regularizer](https://keras.io/regularizers/)).
* **bias\_regularizer**: Regularizer function applied to the bias vector (see [regularizer](https://keras.io/regularizers/)).
* **activity\_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see [regularizer](https://keras.io/regularizers/)).
* **kernel\_constraint**: Constraint function applied to the kernel matrix (see [constraints](https://keras.io/constraints/)).
* **bias\_constraint**: Constraint function applied to the bias vector (see [constraints](https://keras.io/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.

### Conv2D

keras.layers.Conv2D(filters, kernel\_size, strides=(1, 1), padding='valid', data\_format=**None**, dilation\_rate=(1, 1), activation=**None**, use\_bias=**True**, kernel\_initializer='glorot\_uniform', bias\_initializer='zeros', kernel\_regularizer=**None**, bias\_regularizer=**None**, activity\_regularizer=**None**, kernel\_constraint=**None**, bias\_constraint=**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 withstrides != 1, as described [here](https://github.com/keras-team/keras/pull/9473#issuecomment-372166860)
* **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\_firstcorresponds 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](https://keras.io/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](https://keras.io/initializers/)).
* **bias\_initializer**: Initializer for the bias vector (see [initializers](https://keras.io/initializers/)).
* **kernel\_regularizer**: Regularizer function applied to the kernel weights matrix (see [regularizer](https://keras.io/regularizers/)).
* **bias\_regularizer**: Regularizer function applied to the bias vector (see [regularizer](https://keras.io/regularizers/)).
* **activity\_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see [regularizer](https://keras.io/regularizers/)).
* **kernel\_constraint**: Constraint function applied to the kernel matrix (see [constraints](https://keras.io/constraints/)).
* **bias\_constraint**: Constraint function applied to the bias vector (see [constraints](https://keras.io/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.

### SeparableConv2D

keras.layers.SeparableConv2D(filters, kernel\_size, strides=(1, 1), padding='valid', data\_format=**None**, dilation\_rate=(1, 1), depth\_multiplier=1, activation=**None**, use\_bias=**True**, depthwise\_initializer='glorot\_uniform', pointwise\_initializer='glorot\_uniform', bias\_initializer='zeros', depthwise\_regularizer=**None**, pointwise\_regularizer=**None**, bias\_regularizer=**None**, activity\_regularizer=**None**, depthwise\_constraint=**None**, pointwise\_constraint=**None**, bias\_constraint=**None**)

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\_firstcorresponds 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](https://keras.io/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](https://keras.io/initializers/)).
* **pointwise\_initializer**: Initializer for the pointwise kernel matrix (see [initializers](https://keras.io/initializers/)).
* **bias\_initializer**: Initializer for the bias vector (see [initializers](https://keras.io/initializers/)).
* **depthwise\_regularizer**: Regularizer function applied to the depthwise kernel matrix (see [regularizer](https://keras.io/regularizers/)).
* **pointwise\_regularizer**: Regularizer function applied to the pointwise kernel matrix (see [regularizer](https://keras.io/regularizers/)).
* **bias\_regularizer**: Regularizer function applied to the bias vector (see [regularizer](https://keras.io/regularizers/)).
* **activity\_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see [regularizer](https://keras.io/regularizers/)).
* **depthwise\_constraint**: Constraint function applied to the depthwise kernel matrix (see [constraints](https://keras.io/constraints/)).
* **pointwise\_constraint**: Constraint function applied to the pointwise kernel matrix (see [constraints](https://keras.io/constraints/)).
* **bias\_constraint**: Constraint function applied to the bias vector (see [constraints](https://keras.io/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.

### Conv2DTranspose

keras.layers.Conv2DTranspose(filters, kernel\_size, strides=(1, 1), padding='valid', data\_format=**None**, activation=**None**, use\_bias=**True**, kernel\_initializer='glorot\_uniform', bias\_initializer='zeros', kernel\_regularizer=**None**, bias\_regularizer=**None**, activity\_regularizer=**None**, kernel\_constraint=**None**, bias\_constraint=**None**)

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\_firstcorresponds 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](https://keras.io/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](https://keras.io/initializers/)).
* **bias\_initializer**: Initializer for the bias vector (see [initializers](https://keras.io/initializers/)).
* **kernel\_regularizer**: Regularizer function applied to the kernel weights matrix (see [regularizer](https://keras.io/regularizers/)).
* **bias\_regularizer**: Regularizer function applied to the bias vector (see [regularizer](https://keras.io/regularizers/)).
* **activity\_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see [regularizer](https://keras.io/regularizers/)).
* **kernel\_constraint**: Constraint function applied to the kernel matrix (see [constraints](https://keras.io/constraints/)).
* **bias\_constraint**: Constraint function applied to the bias vector (see [constraints](https://keras.io/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](https://arxiv.org/abs/1603.07285v1)
* [Deconvolutional Networks](http://www.matthewzeiler.com/pubs/cvpr2010/cvpr2010.pdf)

### Conv3D

keras.layers.Conv3D(filters, kernel\_size, strides=(1, 1, 1), padding='valid', data\_format=**None**, dilation\_rate=(1, 1, 1), activation=**None**, use\_bias=**True**, kernel\_initializer='glorot\_uniform', bias\_initializer='zeros', kernel\_regularizer=**None**, bias\_regularizer=**None**, activity\_regularizer=**None**, kernel\_constraint=**None**, bias\_constraint=**None**)

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](https://keras.io/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](https://keras.io/initializers/)).
* **bias\_initializer**: Initializer for the bias vector (see [initializers](https://keras.io/initializers/)).
* **kernel\_regularizer**: Regularizer function applied to the kernel weights matrix (see [regularizer](https://keras.io/regularizers/)).
* **bias\_regularizer**: Regularizer function applied to the bias vector (see [regularizer](https://keras.io/regularizers/)).
* **activity\_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see [regularizer](https://keras.io/regularizers/)).
* **kernel\_constraint**: Constraint function applied to the kernel matrix (see [constraints](https://keras.io/constraints/)).
* **bias\_constraint**: Constraint function applied to the bias vector (see [constraints](https://keras.io/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.

### 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)

### 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\_firstcorresponds 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\_formatis "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)*

### 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)

### 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).

### 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\_firstcorresponds 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\_formatis "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)

### 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)

### 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)

### 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\_firstcorresponds 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\_formatis "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)

### 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:

### 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).

### 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\_firstcorresponds 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)

### MaxPooling3D

keras.layers.MaxPooling3D(pool\_size=(2, 2, 2), strides=**None**, padding='valid', data\_format=**None**)

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)

### 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).

### AveragePooling2D

keras.layers.AveragePooling2D(pool\_size=(2, 2), strides=**None**, padding='valid', data\_format=**None**)

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\_firstcorresponds 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)

### AveragePooling3D

keras.layers.AveragePooling3D(pool\_size=(2, 2, 2), strides=**None**, padding='valid', data\_format=**None**)

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)

### 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)

### 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)

### 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\_firstcorresponds 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)

### 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\_firstcorresponds 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).

**Image preprocessing:**

## ImageDataGenerator class

keras.preprocessing.image.ImageDataGenerator(featurewise\_center=**False**, samplewise\_center=**False**, featurewise\_std\_normalization=**False**, samplewise\_std\_normalization=**False**, zca\_whitening=**False**, zca\_epsilon=1e-06, rotation\_range=0.0, width\_shift\_range=0.0, height\_shift\_range=0.0, brightness\_range=**None**, shear\_range=0.0, zoom\_range=0.0, channel\_shift\_range=0.0, fill\_mode='nearest', cval=0.0, horizontal\_flip=**False**, vertical\_flip=**False**, rescale=**None**, preprocessing\_function=**None**, data\_format=**None**, validation\_split=0.0)

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=2possible values are integers [-1, 0, +1], same as with width\_shift\_range=[-1, 0, +1], while with width\_shift\_range=1.0possible 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**: One of {"channels\_first", "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)

## ImageDataGenerator methods

### fit

fit(x, augment=**False**, rounds=1, seed=**None**)

Compute the internal data stats related to the data-dependent transformations, based on an array of sample data. Only required if featurewise\_center or featurewise\_std\_normalization or zca\_whitening.

**Arguments**

* **x**: sample data. Should have rank 4. In case of grayscale data, the channels axis should have value 1, and in case of RGB data, it should have value 3.
* **augment**: Boolean (default: False). Whether to fit on randomly augmented samples.
* **rounds**: int (default: 1). If augment, how many augmentation passes over the data to use.
* **seed**: int (default: None). Random seed.

### flow

flow(x, y=**None**, batch\_size=32, shuffle=**True**, seed=**None**, save\_to\_dir=**None**, save\_prefix='', save\_format='png', subset=**None**)

Takes numpy data & label arrays, and generates batches of augmented/normalized data.

**Arguments**

* **x**: data. Should have rank 4. In case of grayscale data, the channels axis 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 and y is a numpy array of corresponding labels.

### flow\_from\_directory

flow\_from\_directory(directory, target\_size=(256, 256), color\_mode='rgb', classes=**None**, class\_mode='categorical', batch\_size=32, shuffle=**True**, seed=**None**, save\_to\_dir=**None**, save\_prefix='', save\_format='png', follow\_links=**False**, subset=**None**, interpolation='nearest')

Takes the path to a directory, and generates batches of augmented/normalized data.

**Arguments**

* **directory**: path to the target directory. It should contain one subdirectory per class. Any PNG, JPG, BMP, PPM or TIF images inside each of the subdirectories directory tree will be included in the generator. See [this script](https://gist.github.com/fchollet/0830affa1f7f19fd47b06d4cf89ed44d) for more details.
* **target\_size**: tuple of integers (height, width), default: (256, 256). The dimensions to which all images found will be resized.
* **color\_mode**: one of "grayscale", "rbg". Default: "rgb". Whether the images will be converted to have 1 or 3 color channels.
* **classes**: optional list of class subdirectories (e.g. ['dogs', 'cats']). Default: None. If not provided, the list of classes will be automatically inferred from the subdirectory names/structure under directory, where each subdirectory will be treated as a different class (and the order of the classes, which will map to the label indices, will be alphanumeric). The dictionary containing the mapping from class names to class indices can be obtained via the attribute class\_indices.
* **class\_mode**: one of "categorical", "binary", "sparse", "input" or None. Default: "categorical". Determines the type of label arrays that are returned: "categorical" will be 2D one-hot encoded labels, "binary" will be 1D binary labels, "sparse" will be 1D integer labels, "input" will be images identical to input images (mainly used to work with autoencoders). If None, no labels are returned (the generator will only yield batches of image data, which is useful to use model.predict\_generator(), model.evaluate\_generator(), etc.). Please note that in case of class\_mode None, the data still needs to reside in a subdirectory of directory for it to work correctly.
* **batch\_size**: size of the batches of data (default: 32).
* **shuffle**: whether to shuffle the data (default: True)
* **seed**: optional random seed for shuffling and transformations.
* **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. 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".
* **follow\_links**: whether to follow symlinks inside class subdirectories (default: False).
* **subset**: Subset of data ("training" or "validation") if validation\_split is set in ImageDataGenerator.
* **interpolation**: Interpolation method used to resample the image if the target size is different from that of the loaded image. Supported methods are "nearest", "bilinear", and "bicubic". If PIL version 1.1.3 or newer is installed, "lanczos" is also supported. If PIL version 3.4.0 or newer is installed, "box" and "hamming" are also supported. By default, "nearest"is used.

**Returns**

A DirectoryIterator yielding tuples of (x, y) where x is a numpy array containing a batch of images with shape (batch\_size, \*target\_size, channels) and y is a numpy array of corresponding labels.

### random\_transform

random\_transform(x, seed=**None**)

Randomly augment a single image tensor.

**Arguments**

* **x**: 3D tensor, single image.
* **seed**: random seed.

**Returns**

A randomly transformed version of the input (same shape).

### standardize

standardize(x)

Apply the normalization configuration to a batch of inputs.

**Arguments**

* **x**: batch of inputs to be normalized.

**Returns**

The inputs, normalized.