**Keras Tutorial**

Keras (κέρας) means horn in Greek. It is a reference to a literary image from ancient Greek and Latin literature, first found in the Odyssey. Keras was initially developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System).

**Property of Keras**

* user freindly
* Easy extensibility.
* Modularity

**Installation**

sudo pip3 install keras

you need to have tensorflow installed before that. Its working with python 2 as well but because I have python 3 with tensor flow, I used pip3.

Usage

Core data --> model --> sequence of layers --> simplest one is sequential model

from keras.models import Sequential

for adding layers to DNN

from keras.layers import Dense, Activation

model =Sequential()

model = Sequential([

Dense(32, input\_shape=(784,)),

Activation('relu'),

Dense(10),

Activation('softmax'),

])

model.add(Dense(unit=64,input\_dim=100))

model.add(Activation(‘relu’))

model.add(Activation(‘softmax’))

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

model.compile(loss=keras.losses.categorical\_crossentropy,

optimizer=keras.optimizers.SGD(lr=0.01, momentum=0.9, nesterov=True))

# For a multi-class classification problem

model.compile(optimizer='rmsprop',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# For a binary classification problem

model.compile(optimizer='rmsprop',

loss='binary\_crossentropy',

metrics=['accuracy'])

# For a mean squared error regression problem

model.compile(optimizer='rmsprop',

loss='mse')

**# For custom metrics**

import keras.backend as K

def mean\_pred(y\_true, y\_pred):

return K.mean(y\_pred)

**You can now iterate on your training data in batches:**

# x\_train and y\_train are Numpy arrays --just like in the Scikit-Learn API.

model.fit(x\_train, y\_train, epochs=5, batch\_size=32)

Alternatively, you can feed batches to your model manually:

model.train\_on\_batch(x\_batch, y\_batch)

**Evaluate your performance in one line:**

loss\_and\_metrics = model.evaluate(x\_test, y\_test, batch\_size=128)

Or generate predictions on new data:

classes = model.predict(x\_test, batch\_size=128)

**for converting the labels to one hat labels**

one\_hat\_labels = keras.utils.to\_categoriacal(labels,num\_classes=10)

**\*\*\*\* Specifying the input shape**

* 1D input\_shape=(n784 or None)
* 2D --> Dense(batch\_size, input\_dim=784)
* 3D --> Dense(batch\_size, input\_dim =784 or input\_length=784)

Necessary things to add:

**from keras.models import Sequential**

**from keras.layers.core import Dense, Dropout, Activation, Flatten**

**from keras.layers.convolutional import Convolution3D, MaxPooling3D**

**from keras.optimizers import SGD, RMSprop**

**from keras.utils import np\_utils, generic\_utils**

**model.summary()**

**its an useful function for getting the model structure!**

* **Core Layer**

1. **Dense**

keras.layers.core.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)

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).

* + 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).

**# 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))

1. **Activation**

keras.layers.core.Activation(activation)

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.

1. **Drop out**

keras.layers.core.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.

* 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).
* rate: float between 0 and 1. Fraction of the input units to drop.
* **seed**: A Python integer to use as random seed.

1. **Flatten**

keras.layers.core.Flatten()

model.add(Flatten())

1. **Reshape**

keras.layers.core.Reshape(target\_shape) -->tuple of integers. Does not include the batch axis.

# 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)

1. **permute**

keras.layers.core.Permute(dims)

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

**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.

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

model = Sequential()

model.add(Permute((2, 1), input\_shape=(10, 64)))

# now: model.output\_shape == (None, 64, 10)

# note: `None` is the batch dimension

1. **RepeatVector**
2. **Lambda**
3. **ActivityRegularization**
4. **Masking**

* **Convolutional Layers**

keras.layers.convolutional.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)

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.

1. **CNN2D**

keras.layers.convolutional.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)

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".

***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.

1. **CNN3D**

keras.layers.convolutional.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)

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 output of 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.convolutional.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)

### SeparableConv2D

keras.layers.convolutional.SeparableConv2D(filters, kernel\_size, strides=(1, 1), padding='valid', data\_format=None, 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

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 depthwise 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/)).

I understand some stuff today

* verbose = 0 or 1 could be use for showing the progress bar of training and testing
* metric =[‘accuracy’] had some problem and I just removed that and it does work so, we dont need that.

Batch Normalization

Batch Normalization is a method to reduce internal covariate shift in neural networks, first described in [(1)](https://wiki.tum.de/display/lfdv/Batch+Normalization" \l "BatchNormalization-[1]BatchNormalization), leading to the possible usage of higher learning rates. In principle, the method adds an additional step between the layers, in which the output of the layer before is normalized.

Data Augmentation

Keras provides the [ImageDataGenerator](http://keras.io/preprocessing/image/) class that defines the configuration for image data preparation and augmentation. This includes capabilities such as:

* Sample-wise standardization.
* Feature-wise standardization.
* ZCA whitening.
* Random rotation, shifts, shear and flips.
* Dimension reordering.
* Save augmented images to disk.

An augmented image generator can be created as follows:

**datagen = ImageDataGenerator()**

Rather than performing the operations on your entire image dataset in memory, the API is designed to be iterated by the deep learning model fitting process, creating augmented image data for you just-in-time. This reduces your memory overhead, but adds some additional time cost during model training.

After you have created and configured your ImageDataGenerator, you must fit it on your data. This will calculate any statistics required to actually perform the transforms to your image data. You can do this by calling the fit() function on the data generator and pass it your training dataset.

**datagen.fit(train)**

The data generator itself is in fact an iterator, returning batches of image samples when requested. We can configure the batch size and prepare the data generator and get batches of images by calling the flow() function.

**X\_batch, y\_batch = datagen.flow(train, train, batch\_size=32)**

Finally we can make use of the data generator. Instead of calling the fit() function on our model, we must call the fit\_generator() function and pass in the data generator and the desired length of an epoch as well as the total number of epochs on which to train.

**fit\_generator(datagen, samples\_per\_epoch=len(train), epochs=100)**