[**Keras:基于Theano和TensorFlow的深度学习库**](http://www.cnblogs.com/LittleHann/p/6442161.html)

**catalogue**

[复制代码](javascript:void(0);)

1. 引言

2. 一些基本概念

3. Sequential模型

4. 泛型模型

5. 常用层

6. 卷积层

7. 池化层

8. 递归层Recurrent

9. 嵌入层 Embedding

[复制代码](javascript:void(0);)

**1. 引言**

Keras是一个高层神经网络库，Keras由纯Python编写而成并基Tensorflow或Theano

简易和快速的原型设计（keras具有高度模块化，极简，和可扩充特性）

支持CNN和RNN，或二者的结合

支持任意的链接方案（包括多输入和多输出训练）

无缝CPU和GPU切换

***0x1: Keras设计原则***

1. 模块性: 模型可理解为一个独立的序列或图，完全可配置的模块以最少的代价自由组合在一起。具体而言，网络层、损失函数、优化器、初始化策略、激活函数、正则化方法都是独立的模块，我们可以使用它们来构建自己的模型

2. 极简主义: 每个模块都应该尽量的简洁。每一段代码都应该在初次阅读时都显得直观易懂。没有黑魔法，因为它将给迭代和创新带来麻烦

3. 易扩展性: 添加新模块超级简单的容易，只需要仿照现有的模块编写新的类或函数即可。创建新模块的便利性使得Keras更适合于先进的研究工作

4. 与Python协作: Keras没有单独的模型配置文件类型，模型由python代码描述，使其更紧凑和更易debug，并提供了扩展的便利性

***0x2: 快速开始***

sudo apt-get install libblas-dev liblapack-dev libatlas-base-dev gfortran

pip install scipy

Keras的核心数据结构是“模型”，模型是一种组织网络层的方式。Keras中主要的模型是Sequential模型，Sequential是一系列网络层按顺序构成的栈

from keras.models import Sequential

model = Sequential()

将一些网络层通过.add()堆叠起来，就构成了一个模型：

from keras.layers import Dense, Activation

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

model.add(Activation("relu"))

model.add(Dense(output\_dim=10))

model.add(Activation("softmax"))

完成模型的搭建后，我们需要使用.compile()方法来编译模型：

model.compile(loss='categorical\_crossentropy', optimizer='sgd', metrics=['accuracy'])

编译模型时必须指明损失函数和优化器，如果你需要的话，也可以自己定制损失函数。Keras的一个核心理念就是简明易用同时，保证用户对Keras的绝对控制力度，用户可以根据自己的需要定制自己的模型、网络层，甚至修改源代码

from keras.optimizers import SGD

model.compile(loss='categorical\_crossentropy', optimizer=SGD(lr=0.01, momentum=0.9, nesterov=True))

完成模型编译后，我们在训练数据上按batch进行一定次数的迭代训练，以拟合网络

model.fit(X\_train, Y\_train, nb\_epoch=5, batch\_size=32)

当然，我们也可以手动将一个个batch的数据送入网络中训练，这时候需要使用

model.train\_on\_batch(X\_batch, Y\_batch)

随后，我们可以使用一行代码对我们的模型进行评估，看看模型的指标是否满足我们的要求

loss\_and\_metrics = model.evaluate(X\_test, Y\_test, batch\_size=32)

或者，我们可以使用我们的模型，对新的数据进行预测

classes = model.predict\_classes(X\_test, batch\_size=32)

proba = model.predict\_proba(X\_test, batch\_size=32)

**Relevant Link:**

https://github.com/fchollet/keras

http://playground.tensorflow.org/#activation=tanh&regularization=L1&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0.001&noise=45&networkShape=4,5&seed=0.75320&showTestData=true&discretize=true&percTrainData=50&x=true&y=true&xTimesY=true&xSquared=true&ySquared=true&cosX=false&sinX=true&cosY=false&sinY=true&collectStats=false&problem=classification&initZero=false&hideText=false

**2. 一些基本概念**

***0x1: 符号计算***

Keras的底层库使用Theano或TensorFlow，这两个库也称为Keras的后端。无论是Theano还是TensorFlow，都是一个"符号主义"的库。  
因此，这也使得Keras的编程与传统的Python代码有所差别。笼统的说，符号主义的计算首先定义各种变量，然后建立一个“计算图”，计算图规定了各个变量之间的计算关系。建立好的计算图需要编译已确定其内部细节，然而，此时的计算图还是一个"空壳子"，里面没有任何实际的数据，只有当你把需要运算的输入放进去后，才能在整个模型中形成数据流，从而形成输出值。  
Keras的模型搭建形式就是这种方法，在你搭建Keras模型完毕后，你的模型就是一个空壳子，只有实际生成可调用的函数后(K.function)，输入数据，才会形成真正的数据流

***0x2: 张量***

使用这个词汇的目的是为了表述统一，张量可以看作是向量、矩阵的自然推广，我们用张量来表示广泛的数据类型   
规模最小的张量是0阶张量，即标量，也就是一个数   
当我们把一些数有序的排列起来，就形成了1阶张量，也就是一个向量   
如果我们继续把一组向量有序的排列起来，就形成了2阶张量，也就是一个矩阵   
把矩阵摞起来，就是3阶张量，我们可以称为一个立方体，具有3个颜色通道的彩色图片就是一个这样的立方体  
张量的阶数有时候也称为维度，或者轴，轴这个词翻译自英文axis。譬如一个矩阵[[1,2],[3,4]]，是一个2阶张量，有两个维度或轴，沿着第0个轴（为了与python的计数方式一致，本文档维度和轴从0算起）你看到的是[1,2]，[3,4]两个向量，沿着第1个轴你看到的是[1,3]，[2,4]两个向量。

[复制代码](javascript:void(0);)

import numpy as np

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

sum0 = np.sum(a, axis=0)

sum1 = np.sum(a, axis=1)

print sum0

print sum1

[复制代码](javascript:void(0);)

***0x3: 泛型模型***

在原本的Keras版本中，模型其实有两种

1. 一种叫Sequential，称为序贯模型，也就是单输入单输出，一条路通到底，层与层之间只有相邻关系，跨层连接统统没有。这种模型编译速度快，操作上也比较简单

2. 第二种模型称为Graph，即图模型，这个模型支持多输入多输出，层与层之间想怎么连怎么连，但是编译速度慢。可以看到，Sequential其实是Graph的一个特殊情况

在现在这版Keras中，图模型被移除，而增加了了“functional model API”，这个东西，更加强调了Sequential是特殊情况这一点。一般的模型就称为Model，然后如果你要用简单的Sequential，OK，那还有一个快捷方式Sequential。

**Relevant Link:**

http://keras-cn.readthedocs.io/en/latest/getting\_started/concepts/

**3. Sequential模型**

Sequential是多个网络层的线性堆叠  
可以通过向Sequential模型传递一个layer的list来构造该模型

[复制代码](javascript:void(0);)

from keras.models import Sequential

from keras.layers import Dense, Activation

model = Sequential([

Dense(32, input\_dim=784),

Activation('relu'),

Dense(10),

Activation('softmax'),

])

[复制代码](javascript:void(0);)

也可以通过.add()方法一个个的将layer加入模型中：

model = Sequential()

model.add(Dense(32, input\_dim=784))

model.add(Activation('relu'))

***0x1: 指定输入数据的shape***

模型需要知道输入数据的shape，因此，Sequential的第一层需要接受一个关于输入数据shape的参数，后面的各个层则可以自动的推导出中间数据的shape，因此不需要为每个层都指定这个参数。有几种方法来为第一层指定输入数据的shape

1. 传递一个input\_shape的关键字参数给第一层，input\_shape是一个tuple类型的数据，其中也可以填入None，如果填入None则表示此位置可能是任何正整数。数据的batch大小不应包含在其中。

2. 传递一个batch\_input\_shape的关键字参数给第一层，该参数包含数据的batch大小。该参数在指定固定大小batch时比较有用，例如在stateful RNNs中。事实上，Keras在内部会通过添加一个None将input\_shape转化为batch\_input\_shape

3. 有些2D层，如Dense，支持通过指定其输入维度input\_dim来隐含的指定输入数据shape。一些3D的时域层支持通过参数input\_dim和input\_length来指定输入shape

下面的三个指定输入数据shape的方法是严格等价的

[复制代码](javascript:void(0);)

model = Sequential()

model.add(Dense(32, input\_shape=(784,)))

model = Sequential()

model.add(Dense(32, batch\_input\_shape=(None, 784)))

# note that batch dimension is "None" here,

# so the model will be able to process batches of any size.</pre>

model = Sequential()

model.add(Dense(32, input\_dim=784))

[复制代码](javascript:void(0);)

下面三种方法也是严格等价的：

[复制代码](javascript:void(0);)

model = Sequential()

model.add(LSTM(32, input\_shape=(10, 64)))

model = Sequential()

model.add(LSTM(32, batch\_input\_shape=(None, 10, 64)))

model = Sequential()

model.add(LSTM(32, input\_length=10, input\_dim=64))

[复制代码](javascript:void(0);)

***0x2: Merge层***

多个Sequential可经由一个Merge层合并到一个输出。Merge层的输出是一个可以被添加到新  Sequential的层对象。下面这个例子将两个Sequential合并到一起(activation得到最终结果矩阵)

[复制代码](javascript:void(0);)

from keras.layers import Merge

left\_branch = Sequential()

left\_branch.add(Dense(32, input\_dim=784))

right\_branch = Sequential()

right\_branch.add(Dense(32, input\_dim=784))

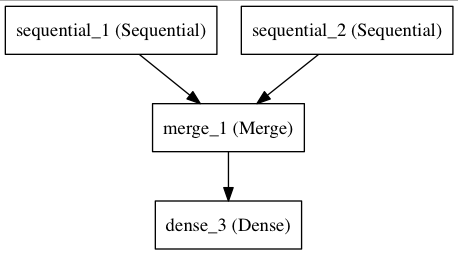
merged = Merge([left\_branch, right\_branch], mode='concat')

final\_model = Sequential()

final\_model.add(merged)

final\_model.add(Dense(10, activation='softmax'))

[复制代码](javascript:void(0);)



Merge层支持一些预定义的合并模式，包括

sum(defualt):逐元素相加

concat:张量串联，可以通过提供concat\_axis的关键字参数指定按照哪个轴进行串联

mul：逐元素相乘

ave：张量平均

dot：张量相乘，可以通过dot\_axis关键字参数来指定要消去的轴

cos：计算2D张量（即矩阵）中各个向量的余弦距离

这个两个分支的模型可以通过下面的代码训练:

final\_model.compile(optimizer='rmsprop', loss='categorical\_crossentropy')

final\_model.fit([input\_data\_1, input\_data\_2], targets) # we pass one data array per model input

也可以为Merge层提供关键字参数mode，以实现任意的变换，例如

merged = Merge([left\_branch, right\_branch], mode=lambda x: x[0] - x[1])

对于不能通过Sequential和Merge组合生成的复杂模型，可以参考泛型模型API

***0x3: 编译***

在训练模型之前，我们需要通过compile来对学习过程进行配置。compile接收三个参数

1. 优化器optimizer：该参数可指定为已预定义的优化器名，如rmsprop、adagrad，或一个Optimizer类的对象

2. 损失函数loss：该参数为模型试图最小化的目标函数，它可为预定义的损失函数名，如categorical\_crossentropy、mse，也可以为一个损失函数

3. 指标列表metrics：对分类问题，我们一般将该列表设置为metrics=['accuracy']。指标可以是一个预定义指标的名字,也可以是一个用户定制的函数.指标函数应该返回单个张量,或一个完成metric\_name - > metric\_value映射的字典

指标列表就是用来生成最后的判断结果的

[复制代码](javascript:void(0);)

# 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 metrices

# for custom metrics

import keras.backend as K

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

return K.mean(y\_pred)

def false\_rates(y\_true, y\_pred):

false\_neg = ...

false\_pos = ...

return {

'false\_neg': false\_neg,

'false\_pos': false\_pos,

}

model.compile(optimizer='rmsprop',

loss='binary\_crossentropy',

metrics=['accuracy', mean\_pred, false\_rates])

[复制代码](javascript:void(0);)

***0x4: 训练***

Keras以Numpy数组作为输入数据和标签的数据类型。训练模型一般使用fit函数

[复制代码](javascript:void(0);)

# for a single-input model with 2 classes (binary):

model = Sequential()

model.add(Dense(1, input\_dim=784, activation='sigmoid'))

model.compile(optimizer='rmsprop',

loss='binary\_crossentropy',

metrics=['accuracy'])

# generate dummy data

import numpy as np

data = np.random.random((1000, 784))

labels = np.random.randint(2, size=(1000, 1))

# train the model, iterating on the data in batches

# of 32 samples

model.fit(data, labels, nb\_epoch=10, batch\_size=32)

[复制代码](javascript:void(0);)

另一个栗子

[复制代码](javascript:void(0);)

# for a multi-input model with 10 classes:

left\_branch = Sequential()

left\_branch.add(Dense(32, input\_dim=784))

right\_branch = Sequential()

right\_branch.add(Dense(32, input\_dim=784))

merged = Merge([left\_branch, right\_branch], mode='concat')

model = Sequential()

model.add(merged)

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

model.compile(optimizer='rmsprop',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# generate dummy data

import numpy as np

from keras.utils.np\_utils import to\_categorical

data\_1 = np.random.random((1000, 784))

data\_2 = np.random.random((1000, 784))

# these are integers between 0 and 9

labels = np.random.randint(10, size=(1000, 1))

# we convert the labels to a binary matrix of size (1000, 10)

# for use with categorical\_crossentropy

labels = to\_categorical(labels, 10)

# train the model

# note that we are passing a list of Numpy arrays as training data

# since the model has 2 inputs

model.fit([data\_1, data\_2], labels, nb\_epoch=10, batch\_size=32)

[复制代码](javascript:void(0);)

***0x5: 一些栗子***

**1. 基于多层感知器的softmax多分类**

[复制代码](javascript:void(0);)

from keras.models import Sequential

from keras.layers import Dense, Dropout, Activation

from keras.optimizers import SGD

model = Sequential()

# Dense(64) is a fully-connected layer with 64 hidden units.

# in the first layer, you must specify the expected input data shape:

# here, 20-dimensional vectors.

model.add(Dense(64, input\_dim=20, init='uniform'))

model.add(Activation('tanh'))

model.add(Dropout(0.5))

model.add(Dense(64, init='uniform'))

model.add(Activation('tanh'))

model.add(Dropout(0.5))

model.add(Dense(10, init='uniform'))

model.add(Activation('softmax'))

sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)

model.compile(loss='categorical\_crossentropy',

optimizer=sgd,

metrics=['accuracy'])

model.fit(X\_train, y\_train,

nb\_epoch=20,

batch\_size=16)

score = model.evaluate(X\_test, y\_test, batch\_size=16)

[复制代码](javascript:void(0);)

**2. 相似MLP的另一种实现**

[复制代码](javascript:void(0);)

model = Sequential()

model.add(Dense(64, input\_dim=20, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(64, activation='relu'))

model.add(Dropout(0.5))

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

model.compile(loss='categorical\_crossentropy',

optimizer='adadelta',

metrics=['accuracy'])

[复制代码](javascript:void(0);)

**3. 用于二分类的多层感知器**

[复制代码](javascript:void(0);)

model = Sequential()

model.add(Dense(64, input\_dim=20, init='uniform', activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(64, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary\_crossentropy',

optimizer='rmsprop',

metrics=['accuracy'])

[复制代码](javascript:void(0);)

**4. 类似VGG的卷积神经网络**

[复制代码](javascript:void(0);)

from keras.models import Sequential

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

from keras.layers import Convolution2D, MaxPooling2D

from keras.optimizers import SGD

model = Sequential()

# input: 100x100 images with 3 channels -> (3, 100, 100) tensors.

# this applies 32 convolution filters of size 3x3 each.

model.add(Convolution2D(32, 3, 3, border\_mode='valid', input\_shape=(3, 100, 100)))

model.add(Activation('relu'))

model.add(Convolution2D(32, 3, 3))

model.add(Activation('relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Convolution2D(64, 3, 3, border\_mode='valid'))

model.add(Activation('relu'))

model.add(Convolution2D(64, 3, 3))

model.add(Activation('relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Flatten())

# Note: Keras does automatic shape inference.

model.add(Dense(256))

model.add(Activation('relu'))

model.add(Dropout(0.5))

model.add(Dense(10))

model.add(Activation('softmax'))

sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)

model.compile(loss='categorical\_crossentropy', optimizer=sgd)

model.fit(X\_train, Y\_train, batch\_size=32, nb\_epoch=1)

[复制代码](javascript:void(0);)

**5. 使用LSTM的序列分类**

[复制代码](javascript:void(0);)

from keras.models import Sequential

from keras.layers import Dense, Dropout, Activation

from keras.layers import Embedding

from keras.layers import LSTM

model = Sequential()

model.add(Embedding(max\_features, 256, input\_length=maxlen))

model.add(LSTM(output\_dim=128, activation='sigmoid', inner\_activation='hard\_sigmoid'))

model.add(Dropout(0.5))

model.add(Dense(1))

model.add(Activation('sigmoid'))

model.compile(loss='binary\_crossentropy',

optimizer='rmsprop',

metrics=['accuracy'])

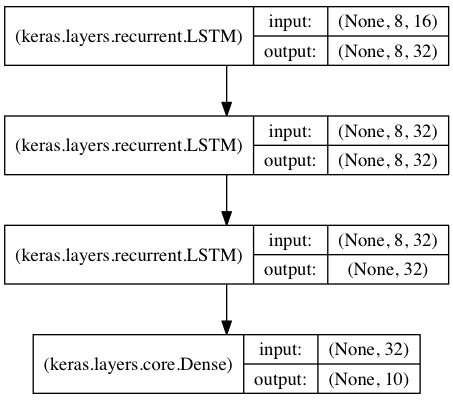
model.fit(X\_train, Y\_train, batch\_size=16, nb\_epoch=10)

score = model.evaluate(X\_test, Y\_test, batch\_size=16)

[复制代码](javascript:void(0);)

**6. 用于序列分类的栈式LSTM**

在该模型中，我们将三个LSTM堆叠在一起，是该模型能够学习更高层次的时域特征表示。  
开始的两层LSTM返回其全部输出序列，而第三层LSTM只返回其输出序列的最后一步结果，从而其时域维度降低（即将输入序列转换为单个向量）



[复制代码](javascript:void(0);)

from keras.models import Sequential

from keras.layers import LSTM, Dense

import numpy as np

data\_dim = 16

timesteps = 8

nb\_classes = 10

# expected input data shape: (batch\_size, timesteps, data\_dim)

model = Sequential()

model.add(LSTM(32, return\_sequences=True,

input\_shape=(timesteps, data\_dim))) # returns a sequence of vectors of dimension 32

model.add(LSTM(32, return\_sequences=True)) # returns a sequence of vectors of dimension 32

model.add(LSTM(32)) # return a single vector of dimension 32

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

model.compile(loss='categorical\_crossentropy',

optimizer='rmsprop',

metrics=['accuracy'])

# generate dummy training data

x\_train = np.random.random((1000, timesteps, data\_dim))

y\_train = np.random.random((1000, nb\_classes))

# generate dummy validation data

x\_val = np.random.random((100, timesteps, data\_dim))

y\_val = np.random.random((100, nb\_classes))

model.fit(x\_train, y\_train,

batch\_size=64, nb\_epoch=5,

validation\_data=(x\_val, y\_val))

[复制代码](javascript:void(0);)

**7. 采用状态LSTM的相同模型**

状态（stateful）LSTM的特点是，在处理过一个batch的训练数据后，其内部状态（记忆）会被作为下一个batch的训练数据的初始状态。状态LSTM使得我们可以在合理的计算复杂度内处理较长序列

[复制代码](javascript:void(0);)

from keras.models import Sequential

from keras.layers import LSTM, Dense

import numpy as np

data\_dim = 16

timesteps = 8

nb\_classes = 10

batch\_size = 32

# expected input batch shape: (batch\_size, timesteps, data\_dim)

# note that we have to provide the full batch\_input\_shape since the network is stateful.

# the sample of index i in batch k is the follow-up for the sample i in batch k-1.

model = Sequential()

model.add(LSTM(32, return\_sequences=True, stateful=True,

batch\_input\_shape=(batch\_size, timesteps, data\_dim)))

model.add(LSTM(32, return\_sequences=True, stateful=True))

model.add(LSTM(32, stateful=True))

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

model.compile(loss='categorical\_crossentropy',

optimizer='rmsprop',

metrics=['accuracy'])

# generate dummy training data

x\_train = np.random.random((batch\_size \* 10, timesteps, data\_dim))

y\_train = np.random.random((batch\_size \* 10, nb\_classes))

# generate dummy validation data

x\_val = np.random.random((batch\_size \* 3, timesteps, data\_dim))

y\_val = np.random.random((batch\_size \* 3, nb\_classes))

model.fit(x\_train, y\_train,

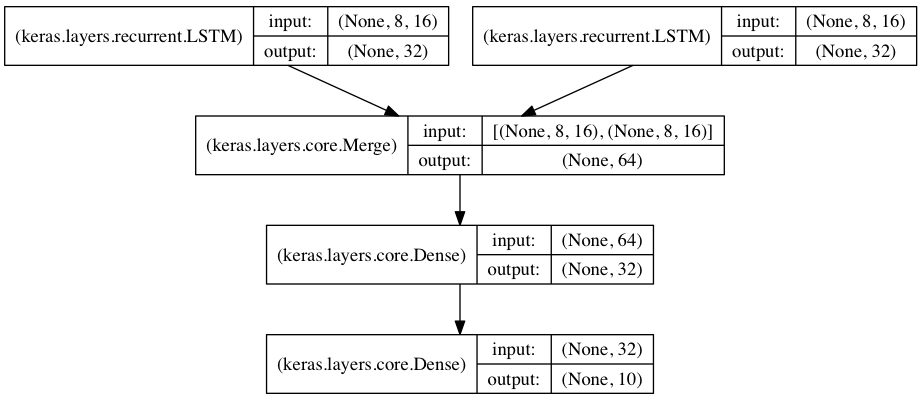
batch\_size=batch\_size, nb\_epoch=5,

validation\_data=(x\_val, y\_val))

[复制代码](javascript:void(0);)

**8. 将两个LSTM合并作为编码端来处理两路序列的分类**

两路输入序列通过两个LSTM被编码为特征向量  
两路特征向量被串连在一起，然后通过一个全连接网络得到结果



[复制代码](javascript:void(0);)

from keras.models import Sequential

from keras.layers import Merge, LSTM, Dense

import numpy as np

data\_dim = 16

timesteps = 8

nb\_classes = 10

encoder\_a = Sequential()

encoder\_a.add(LSTM(32, input\_shape=(timesteps, data\_dim)))

encoder\_b = Sequential()

encoder\_b.add(LSTM(32, input\_shape=(timesteps, data\_dim)))

decoder = Sequential()

decoder.add(Merge([encoder\_a, encoder\_b], mode='concat'))

decoder.add(Dense(32, activation='relu'))

decoder.add(Dense(nb\_classes, activation='softmax'))

decoder.compile(loss='categorical\_crossentropy',

optimizer='rmsprop',

metrics=['accuracy'])

# generate dummy training data

x\_train\_a = np.random.random((1000, timesteps, data\_dim))

x\_train\_b = np.random.random((1000, timesteps, data\_dim))

y\_train = np.random.random((1000, nb\_classes))

# generate dummy validation data

x\_val\_a = np.random.random((100, timesteps, data\_dim))

x\_val\_b = np.random.random((100, timesteps, data\_dim))

y\_val = np.random.random((100, nb\_classes))

decoder.fit([x\_train\_a, x\_train\_b], y\_train,

batch\_size=64, nb\_epoch=5,

validation\_data=([x\_val\_a, x\_val\_b], y\_val))

[复制代码](javascript:void(0);)

**Relevant Link:**

http://www.jianshu.com/p/9dc9f41f0b29

http://keras-cn.readthedocs.io/en/latest/getting\_started/sequential\_model/

**4. 泛型模型**

Keras泛型模型接口是用户定义多输出模型、非循环有向模型或具有共享层的模型等复杂模型的途径

1. 层对象接受张量为参数，返回一个张量。张量在数学上只是数据结构的扩充，一阶张量就是向量，二阶张量就是矩阵，三阶张量就是立方体。在这里张量只是广义的表达一种数据结构，例如一张彩色图像其实就是一个三阶张量(每一阶都是one-hot向量)，它由三个通道的像素值堆叠而成。而10000张彩色图构成的一个数据集合则是四阶张量。

2. 输入是张量，输出也是张量的一个框架就是一个模型

3. 这样的模型可以被像Keras的Sequential一样被训练

例如这个全连接网络

[复制代码](javascript:void(0);)

from keras.layers import Input, Dense

from keras.models import Model

# this returns a tensor

inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor

x = Dense(64, activation='relu')(inputs)

x = Dense(64, activation='relu')(x)

predictions = Dense(10, activation='softmax')(x)

# this creates a model that includes

# the Input layer and three Dense layers

model = Model(input=inputs, output=predictions)

model.compile(optimizer='rmsprop',

loss='categorical\_crossentropy',

metrics=['accuracy'])

model.fit(data, labels) # starts training

[复制代码](javascript:void(0);)

***0x1: 所有的模型都是可调用的，就像层一样***

利用泛型模型的接口，我们可以很容易的重用已经训练好的模型：你可以把模型当作一个层一样，通过提供一个tensor来调用它。注意当你调用一个模型时，你不仅仅重用了它的结构，也重用了它的权重

x = Input(shape=(784,))

# this works, and returns the 10-way softmax we defined above.

y = model(x)

这种方式可以允许你快速的创建能处理序列信号的模型，你可以很快将一个图像分类的模型变为一个对视频分类的模型，只需要一行代码：

[复制代码](javascript:void(0);)

from keras.layers import TimeDistributed

# input tensor for sequences of 20 timesteps,

# each containing a 784-dimensional vector

input\_sequences = Input(shape=(20, 784))

# this applies our previous model to every timestep in the input sequences.

# the output of the previous model was a 10-way softmax,

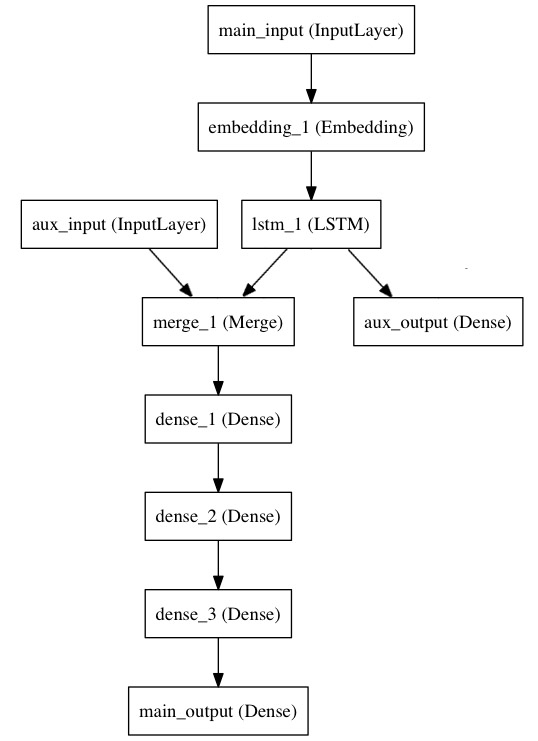
# so the output of the layer below will be a sequence of 20 vectors of size 10.

processed\_sequences = TimeDistributed(model)(input\_sequences)

[复制代码](javascript:void(0);)

***0x2: 多输入和多输出模型***

使用泛型模型的一个典型场景是搭建多输入、多输出的模型。  
考虑这样一个模型。我们希望预测Twitter上一条新闻会被转发和点赞多少次。模型的主要输入是新闻本身，也就是一个词语的序列。但我们还可以拥有额外的输入，如新闻发布的日期等。这个模型的损失函数将由两部分组成，辅助的损失函数评估仅仅基于新闻本身做出预测的情况，主损失函数评估基于新闻和额外信息的预测的情况，即使来自主损失函数的梯度发生弥散，来自辅助损失函数的信息也能够训练Embeddding和LSTM层。在模型中早点使用主要的损失函数是对于深度网络的一个良好的正则方法。总而言之，该模型框图如下：



让我们用泛型模型来实现这个框图  
主要的输入接收新闻本身，即一个整数的序列（每个整数编码了一个词）。这些整数位于1到10，000之间（即我们的字典有10，000个词）。这个序列有100个单词

[复制代码](javascript:void(0);)

from keras.layers import Input, Embedding, LSTM, Dense, merge

from keras.models import Model

# headline input: meant to receive sequences of 100 integers, between 1 and 10000.

# note that we can name any layer by passing it a "name" argument.

main\_input = Input(shape=(100,), dtype='int32', name='main\_input')

# this embedding layer will encode the input sequence

# into a sequence of dense 512-dimensional vectors.

x = Embedding(output\_dim=512, input\_dim=10000, input\_length=100)(main\_input)

# a LSTM will transform the vector sequence into a single vector,

# containing information about the entire sequence

lstm\_out = LSTM(32)(x)

[复制代码](javascript:void(0);)

然后，我们插入一个额外的损失，使得即使在主损失很高的情况下，LSTM和Embedding层也可以平滑的训练

auxiliary\_output = Dense(1, activation='sigmoid', name='aux\_output')(lstm\_out)

再然后，我们将LSTM与额外的输入数据串联起来组成输入，送入模型中

[复制代码](javascript:void(0);)

auxiliary\_input = Input(shape=(5,), name='aux\_input')

x = merge([lstm\_out, auxiliary\_input], mode='concat')

# we stack a deep fully-connected network on top

x = Dense(64, activation='relu')(x)

x = Dense(64, activation='relu')(x)

x = Dense(64, activation='relu')(x)

# and finally we add the main logistic regression layer

main\_output = Dense(1, activation='sigmoid', name='main\_output')(x)

[复制代码](javascript:void(0);)

最后，我们定义整个2输入，2输出的模型：

model = Model(input=[main\_input, auxiliary\_input], output=[main\_output, auxiliary\_output])

模型定义完毕，下一步编译模型。我们给额外的损失赋0.2的权重。我们可以通过关键字参数loss\_weights或loss来为不同的输出设置不同的损失函数或权值。这两个参数均可为Python的列表或字典。这里我们给loss传递单个损失函数，这个损失函数会被应用于所有输出上

model.compile(optimizer='rmsprop', loss='binary\_crossentropy',

loss\_weights=[1., 0.2])

编译完成后，我们通过传递训练数据和目标值训练该模型：

model.fit([headline\_data, additional\_data], [labels, labels],

nb\_epoch=50, batch\_size=32)

因为我们输入和输出是被命名过的（在定义时传递了“name”参数），我们也可以用下面的方式编译和训练模型：

[复制代码](javascript:void(0);)

model.compile(optimizer='rmsprop',

loss={'main\_output': 'binary\_crossentropy', 'aux\_output': 'binary\_crossentropy'},

loss\_weights={'main\_output': 1., 'aux\_output': 0.2})

# and trained it via:

model.fit({'main\_input': headline\_data, 'aux\_input': additional\_data},

{'main\_output': labels, 'aux\_output': labels},

nb\_epoch=50, batch\_size=32)

[复制代码](javascript:void(0);)

***0x3: 共享层***

另一个使用泛型模型的场合是使用共享层的时候  
考虑微博数据，我们希望建立模型来判别两条微博是否是来自同一个用户，这个需求同样可以用来判断一个用户的两条微博的相似性。  
一种实现方式是，我们建立一个模型，它分别将两条微博的数据映射到两个特征向量上，然后将特征向量串联并加一个logistic回归层，输出它们来自同一个用户的概率。这种模型的训练数据是一对对的微博。  
因为这个问题是对称的，所以处理第一条微博的模型当然也能重用于处理第二条微博。所以这里我们使用一个共享的LSTM层来进行映射。  
首先，我们将微博的数据转为（140，256）的矩阵，即每条微博有140个字符，每个单词的特征由一个256维的词向量表示，向量的每个元素为1表示某个字符出现，为0表示不出现，这是一个one-hot编码

from keras.layers import Input, LSTM, Dense, merge

from keras.models import Model

tweet\_a = Input(shape=(140, 256))

tweet\_b = Input(shape=(140, 256))

若要对不同的输入共享同一层，就初始化该层一次，然后多次调用它

[复制代码](javascript:void(0);)

# this layer can take as input a matrix

# and will return a vector of size 64

shared\_lstm = LSTM(64)

# when we reuse the same layer instance

# multiple times, the weights of the layer

# are also being reused

# (it is effectively \*the same\* layer)

encoded\_a = shared\_lstm(tweet\_a)

encoded\_b = shared\_lstm(tweet\_b)

# we can then concatenate the two vectors:

merged\_vector = merge([encoded\_a, encoded\_b], mode='concat', concat\_axis=-1)

# and add a logistic regression on top

predictions = Dense(1, activation='sigmoid')(merged\_vector)

# we define a trainable model linking the

# tweet inputs to the predictions

model = Model(input=[tweet\_a, tweet\_b], output=predictions)

model.compile(optimizer='rmsprop',

loss='binary\_crossentropy',

metrics=['accuracy'])

model.fit([data\_a, data\_b], labels, nb\_epoch=10)

[复制代码](javascript:void(0);)

***0x4: 层“节点”的概念***

无论何时，当你在某个输入上调用层时，你就创建了一个新的张量（即该层的输出），同时你也在为这个层增加一个“（计算）节点”。这个节点将输入张量映射为输出张量。当你多次调用该层时，这个层就有了多个节点，其下标分别为0，1，2...

***0x5: 依旧是一些栗子***

**1. inception模型**

[复制代码](javascript:void(0);)

from keras.layers import merge, Convolution2D, MaxPooling2D, Input

input\_img = Input(shape=(3, 256, 256))

tower\_1 = Convolution2D(64, 1, 1, border\_mode='same', activation='relu')(input\_img)

tower\_1 = Convolution2D(64, 3, 3, border\_mode='same', activation='relu')(tower\_1)

tower\_2 = Convolution2D(64, 1, 1, border\_mode='same', activation='relu')(input\_img)

tower\_2 = Convolution2D(64, 5, 5, border\_mode='same', activation='relu')(tower\_2)

tower\_3 = MaxPooling2D((3, 3), strides=(1, 1), border\_mode='same')(input\_img)

tower\_3 = Convolution2D(64, 1, 1, border\_mode='same', activation='relu')(tower\_3)

output = merge([tower\_1, tower\_2, tower\_3], mode='concat', concat\_axis=1)

[复制代码](javascript:void(0);)

**2. 卷积层的残差连接(Residual Network)**

[复制代码](javascript:void(0);)

from keras.layers import merge, Convolution2D, Input

# input tensor for a 3-channel 256x256 image

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

# 3x3 conv with 3 output channels(same as input channels)

y = Convolution2D(3, 3, 3, border\_mode='same')(x)

# this returns x + y.

z = merge([x, y], mode='sum')

[复制代码](javascript:void(0);)

**3. 共享视觉模型**

该模型在两个输入上重用了图像处理的模型，用来判别两个MNIST数字是否是相同的数字

[复制代码](javascript:void(0);)

from keras.layers import merge, Convolution2D, MaxPooling2D, Input, Dense, Flatten

from keras.models import Model

# first, define the vision modules

digit\_input = Input(shape=(1, 27, 27))

x = Convolution2D(64, 3, 3)(digit\_input)

x = Convolution2D(64, 3, 3)(x)

x = MaxPooling2D((2, 2))(x)

out = Flatten()(x)

vision\_model = Model(digit\_input, out)

# then define the tell-digits-apart model

digit\_a = Input(shape=(1, 27, 27))

digit\_b = Input(shape=(1, 27, 27))

# the vision model will be shared, weights and all

out\_a = vision\_model(digit\_a)

out\_b = vision\_model(digit\_b)

concatenated = merge([out\_a, out\_b], mode='concat')

out = Dense(1, activation='sigmoid')(concatenated)

classification\_model = Model([digit\_a, digit\_b], out)

[复制代码](javascript:void(0);)

**4. 视觉问答模型(问题性图像验证码)**

在针对一幅图片使用自然语言进行提问时，该模型能够提供关于该图片的一个单词的答案  
这个模型将自然语言的问题和图片分别映射为特征向量，将二者合并后训练一个logistic回归层，从一系列可能的回答中挑选一个。

[复制代码](javascript:void(0);)

from keras.layers import Convolution2D, MaxPooling2D, Flatten

from keras.layers import Input, LSTM, Embedding, Dense, merge

from keras.models import Model, Sequential

# first, let's define a vision model using a Sequential model.

# this model will encode an image into a vector.

vision\_model = Sequential()

vision\_model.add(Convolution2D(64, 3, 3, activation='relu', border\_mode='same', input\_shape=(3, 224, 224)))

vision\_model.add(Convolution2D(64, 3, 3, activation='relu'))

vision\_model.add(MaxPooling2D((2, 2)))

vision\_model.add(Convolution2D(128, 3, 3, activation='relu', border\_mode='same'))

vision\_model.add(Convolution2D(128, 3, 3, activation='relu'))

vision\_model.add(MaxPooling2D((2, 2)))

vision\_model.add(Convolution2D(256, 3, 3, activation='relu', border\_mode='same'))

vision\_model.add(Convolution2D(256, 3, 3, activation='relu'))

vision\_model.add(Convolution2D(256, 3, 3, activation='relu'))

vision\_model.add(MaxPooling2D((2, 2)))

vision\_model.add(Flatten())

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

image\_input = Input(shape=(3, 224, 224))

encoded\_image = vision\_model(image\_input)

# next, let's define a language model to encode the question into a vector.

# each question will be at most 100 word long,

# and we will index words as integers from 1 to 9999.

question\_input = Input(shape=(100,), dtype='int32')

embedded\_question = Embedding(input\_dim=10000, output\_dim=256, input\_length=100)(question\_input)

encoded\_question = LSTM(256)(embedded\_question)

# let's concatenate the question vector and the image vector:

merged = merge([encoded\_question, encoded\_image], mode='concat')

# and let's train a logistic regression over 1000 words on top:

output = Dense(1000, activation='softmax')(merged)

# this is our final model:

vqa\_model = Model(input=[image\_input, question\_input], output=output)

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

[复制代码](javascript:void(0);)

**5. 视频问答模型**

在做完图片问答模型后，我们可以快速将其转为视频问答的模型。在适当的训练下，你可以为模型提供一个短视频（如100帧）然后向模型提问一个关于该视频的问题，如“what sport is the boy playing？”->“football”

[复制代码](javascript:void(0);)

from keras.layers import TimeDistributed

video\_input = Input(shape=(100, 3, 224, 224))

# this is our video encoded via the previously trained vision\_model (weights are reused)

encoded\_frame\_sequence = TimeDistributed(vision\_model)(video\_input) # the output will be a sequence of vectors

encoded\_video = LSTM(256)(encoded\_frame\_sequence) # the output will be a vector

# this is a model-level representation of the question encoder, reusing the same weights as before:

question\_encoder = Model(input=question\_input, output=encoded\_question)

# let's use it to encode the question:

video\_question\_input = Input(shape=(100,), dtype='int32')

encoded\_video\_question = question\_encoder(video\_question\_input)

# and this is our video question answering model:

merged = merge([encoded\_video, encoded\_video\_question], mode='concat')

output = Dense(1000, activation='softmax')(merged)

video\_qa\_model = Model(input=[video\_input, video\_question\_input], output=output)

[复制代码](javascript:void(0);)

**Relevant Link:**

http://wiki.jikexueyuan.com/project/tensorflow-zh/resources/dims\_types.html

**5. 常用层**

***0x1: Dense层***

Dense就是常用的全连接层

[复制代码](javascript:void(0);)

keras.layers.core.Dense(

output\_dim,

init='glorot\_uniform',

activation='linear',

weights=None,

W\_regularizer=None,

b\_regularizer=None,

activity\_regularizer=None,

W\_constraint=None,

b\_constraint=None,

bias=True,

input\_dim=None

)

1. output\_dim：大于0的整数，代表该层的输出维度。模型中非首层的全连接层其输入维度可以自动推断，因此非首层的全连接定义时不需要指定输入维度。

2. init：初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的Theano函数。该参数仅在不传递weights参数时才有意义。

3. activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）

4. weights：权值，为numpy array的list。该list应含有一个形如（input\_dim,output\_dim）的权重矩阵和一个形如(output\_dim,)的偏置向量。

5. W\_regularizer：施加在权重上的正则项，为WeightRegularizer对象

6. b\_regularizer：施加在偏置向量上的正则项，为WeightRegularizer对象

7. activity\_regularizer：施加在输出上的正则项，为ActivityRegularizer对象

8. W\_constraints：施加在权重上的约束项，为Constraints对象

9. b\_constraints：施加在偏置上的约束项，为Constraints对象

10. bias：布尔值，是否包含偏置向量（即层对输入做线性变换还是仿射变换）

11. input\_dim：整数，输入数据的维度。当Dense层作为网络的第一层时，必须指定该参数或input\_shape参数。

[复制代码](javascript:void(0);)

after the first layer, you don't need to specify the size of the input anymore

***0x2: Activation层***

激活层对一个层的输出施加激活函数

keras.layers.core.Activation(activation)

activation：将要使用的激活函数，为预定义激活函数名或一个Tensorflow/Theano的函数

***0x3: Dropout层***

为输入数据施加Dropout。Dropout将在训练过程中每次更新参数时随机断开一定百分比（p）的输入神经元连接，Dropout层用于防止过拟合

keras.layers.core.Dropout(p)

p：0~1的浮点数，控制需要断开的链接的比例

***0x4: Flatten层***

Flatten层用来将输入“压平”，即把多维的输入一维化，常用在从卷积层到全连接层的过渡。Flatten不影响batch的大小

[复制代码](javascript:void(0);)

keras.layers.core.Flatten()

model = Sequential()

model.add(Convolution2D(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)

[复制代码](javascript:void(0);)

***0x5: Reshape层***

Reshape层用来将输入shape转换为特定的shape

[复制代码](javascript:void(0);)

keras.layers.core.Reshape(target\_shape)

target\_shape：目标shape，为整数的tuple，不包含样本数目的维度（batch大小）

# 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)

[复制代码](javascript:void(0);)

***0x6: Permute层***  
Permute层将输入的维度按照给定模式进行重排，例如，当需要将RNN和CNN网络连接时，可能会用到该层

[复制代码](javascript:void(0);)

keras.layers.core.Permute(dims)

dims：整数tuple，指定重排的模式，不包含样本数的维度。重排模式的下标从1开始。例如（2，1）代表将输入的第二个维度重拍到输出的第一个维度，而将输入的第一个维度重排到第二个维度

model = Sequential()

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

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

# note: `None` is the batch dimension

[复制代码](javascript:void(0);)

***0x7: RepeatVector层***

RepeatVector层将输入重复n次

[复制代码](javascript:void(0);)

keras.layers.core.RepeatVector(n)

n：整数，重复的次数

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)

[复制代码](javascript:void(0);)

***0x8: Merge层***

Merge层根据给定的模式，将一个张量列表中的若干张量合并为一个单独的张量

[复制代码](javascript:void(0);)

keras.engine.topology.Merge(

layers=None,

mode='sum',

concat\_axis=-1,

dot\_axes=-1,

output\_shape=None,

node\_indices=None,

tensor\_indices=None,

name=None

)

1. layers：该参数为Keras张量的列表，或Keras层对象的列表。该列表的元素数目必须大于1。

2. mode：合并模式，为预定义合并模式名的字符串或lambda函数或普通函数，如果为lambda函数或普通函数，则该函数必须接受一个张量的list作为输入，并返回一个张量。如果为字符串，则必须是下列值之一：

“sum”，“mul”，“concat”，“ave”，“cos”，“dot”

3. concat\_axis：整数，当mode=concat时指定需要串联的轴

4. dot\_axes：整数或整数tuple，当mode=dot时，指定要消去的轴

5. output\_shape：整数tuple或lambda函数/普通函数（当mode为函数时）。如果output\_shape是函数时，该函数的输入值应为一一对应于输入shape的list，并返回输出张量的shape。

6. node\_indices：可选，为整数list，如果有些层具有多个输出节点（node）的话，该参数可以指定需要merge的那些节点的下标。如果没有提供，该参数的默认值为全0向量，即合并输入层0号节点的输出值。

7. tensor\_indices：可选，为整数list，如果有些层返回多个输出张量的话，该参数用以指定需要合并的那些张量

[复制代码](javascript:void(0);)

在进行merge的时候需要仔细思考采用哪种连接方式，以及将哪个轴进行merge，因为这会很大程度上影响神经网络的训练过程

***0x9: Lambda层***

本函数用以对上一层的输出施以任何Theano/TensorFlow表达式

[复制代码](javascript:void(0);)

keras.layers.core.Lambda(

function,

output\_shape=None,

arguments={}

)

1. function：要实现的函数，该函数仅接受一个变量，即上一层的输出

2. output\_shape：函数应该返回的值的shape，可以是一个tuple，也可以是一个根据输入shape计算输出shape的函数

3. arguments：可选，字典，用来记录向函数中传递的其他关键字参数

[复制代码](javascript:void(0);)

***0x10: ActivityRegularizer层***

经过本层的数据不会有任何变化，但会基于其激活值更新损失函数值

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

l1：1范数正则因子（正浮点数）

l2：2范数正则因子（正浮点数）

***0x11: Masking层***

使用给定的值对输入的序列信号进行“屏蔽”，用以定位需要跳过的时间步  
对于输入张量的时间步，即输入张量的第1维度（维度从0开始算），如果输入张量在该时间步上都等于mask\_value，则该时间步将在模型接下来的所有层（只要支持masking）被跳过（屏蔽）。  
如果模型接下来的一些层不支持masking，却接受到masking过的数据，则抛出异常

[复制代码](javascript:void(0);)

考虑输入数据x是一个形如(samples,timesteps,features)的张量，现将其送入LSTM层。因为你缺少时间步为3和5的信号，所以你希望将其掩盖。这时候应该：

赋值x[:,3,:] = 0.，x[:,5,:] = 0.

在LSTM层之前插入mask\_value=0.的Masking层

model = Sequential()

model.add(Masking(mask\_value=0., input\_shape=(timesteps, features)))

model.add(LSTM(32))

[复制代码](javascript:void(0);)

***0x12: Highway层***

Highway层建立全连接的Highway网络，这是LSTM在前馈神经网络中的推广

[复制代码](javascript:void(0);)

keras.layers.core.Highway(

init='glorot\_uniform',

transform\_bias=-2,

activation='linear',

weights=None,

W\_regularizer=None,

b\_regularizer=None,

activity\_regularizer=None,

W\_constraint=None,

b\_constraint=None,

bias=True,

input\_dim=None

)

output\_dim：大于0的整数，代表该层的输出维度。模型中非首层的全连接层其输入维度可以自动推断，因此非首层的全连接定义时不需要指定输入维度。

init：初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的Theano函数。该参数仅在不传递weights参数时有意义。

activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）

weights：权值，为numpy array的list。该list应含有一个形如（input\_dim,output\_dim）的权重矩阵和一个形如(output\_dim,)的偏置向量。

W\_regularizer：施加在权重上的正则项，为WeightRegularizer对象

b\_regularizer：施加在偏置向量上的正则项，为WeightRegularizer对象

activity\_regularizer：施加在输出上的正则项，为ActivityRegularizer对象

W\_constraints：施加在权重上的约束项，为Constraints对象

b\_constraints：施加在偏置上的约束项，为Constraints对象

bias：布尔值，是否包含偏置向量（即层对输入做线性变换还是仿射变换）

input\_dim：整数，输入数据的维度。当该层作为网络的第一层时，必须指定该参数或input\_shape参数。

transform\_bias：用以初始化传递参数，默认为-2（请参考文献理解本参数的含义）

[复制代码](javascript:void(0);)

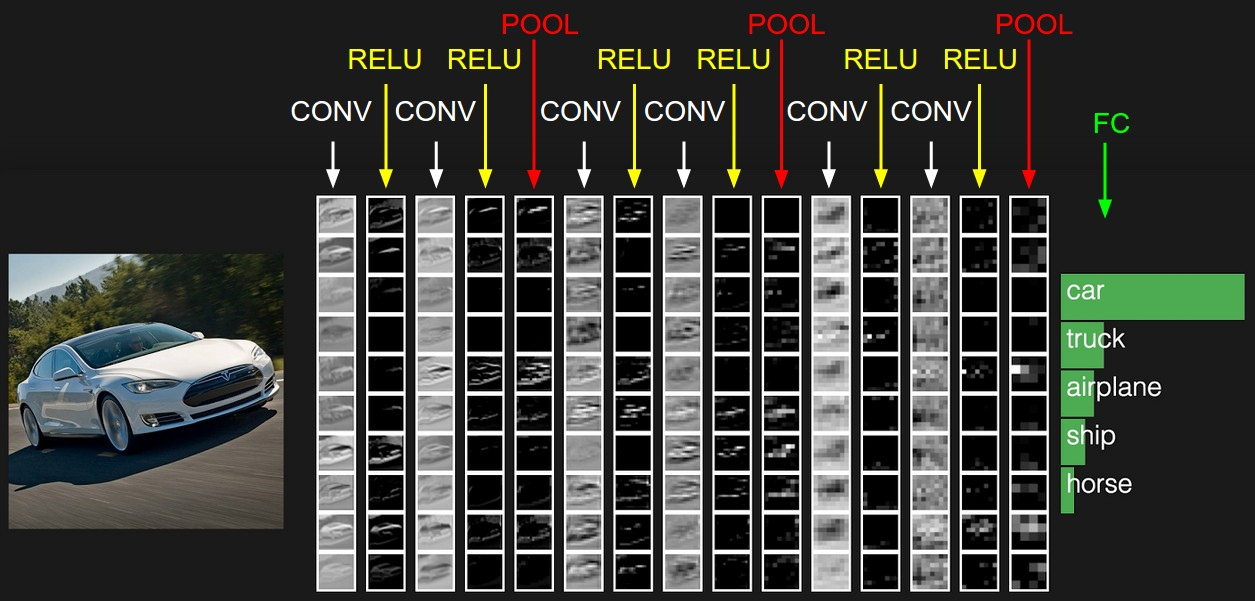
***0x13: MaxoutDense层***

全连接的Maxout层。MaxoutDense层以nb\_features个Dense(input\_dim,output\_dim)线性层的输出的最大值为输出。MaxoutDense可对输入学习出一个凸的、分段线性的激活函数

***Relevant Link:***

https://keras-cn.readthedocs.io/en/latest/layers/core\_layer/

**6. 卷积层**

****

[复制代码](javascript:void(0);)

数据输入层: 对数据做一些处理，比如去均值(把输入数据各个维度都中心化为0，避免数据过多偏差，影响训练效果)、归一化(把所有的数据都归一到同样的范围)、PCA/白化等等

中间是

CONV: 卷积计算层，线性乘积 求和(内积)

RELU: 激励层(激活函数)，用于把向量转化为一个"量值"，用于评估本轮参数的分类效果

POOL: 池化层，简言之，即取区域平均或最大

最右边是

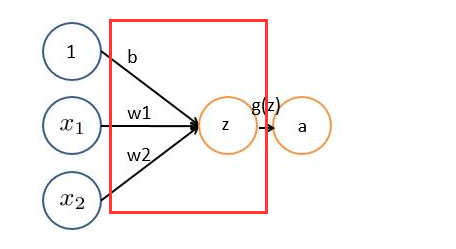
FC: 全连接层

[复制代码](javascript:void(0);)

***0x0: CNN之卷积计算层***

**1. CNN核心概念: 滤波**

在通信领域中，滤波(Wave filtering)指的是将信号中特定波段频率滤除的操作，是抑制和防止干扰的一项重要措施。在CNN图像识别领域，指的是对图像(不同的数据窗口数据)和滤波矩阵(一组固定的权重：因为每个神经元的多个权重固定，所以又可以看做一个恒定的滤波器filter)做内积(逐个元素相乘再求和)的操作就是所谓的"卷积"操作，也是卷积神经网络的名字来源。  
直观上理解就是从一个区域(区域的大小就是filter滤波器的size)中抽取出"重要的细节"，而抽取的方法就是建立"区域权重"，根据区域权重把一个区域中的重点细节过滤出来  
再直观一些理解就是例如上图的汽车图像，滤波器要做的就是把其中的轮胎、车后视镜、前脸轮廓、A柱形状过滤出来，从边缘细节的角度来看待一张非格式化的图像  
这种技术的理论基础是学术界认为人眼对图像的识别也是分层的，人眼第一眼接收到的就是一个物理的轮廓细节，然后传输给大脑皮层，然后在轮廓细节的基础上进一步抽象建立起对一个物理的整体感知

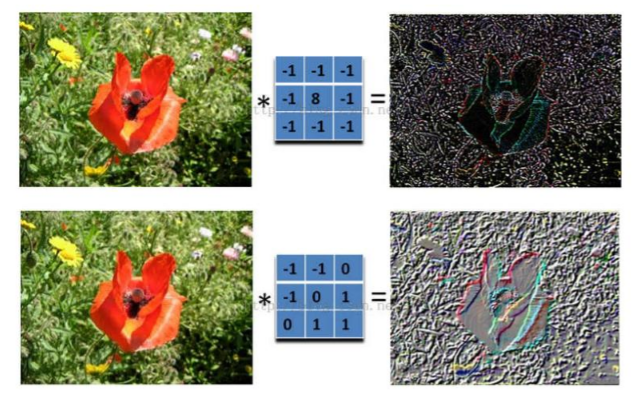


非严格意义上来讲，上图中红框框起来的部分便可以理解为一个滤波器，即带着一组固定权重的神经元。多个滤波器叠加便成了卷积层

**2. 图像上的卷积**

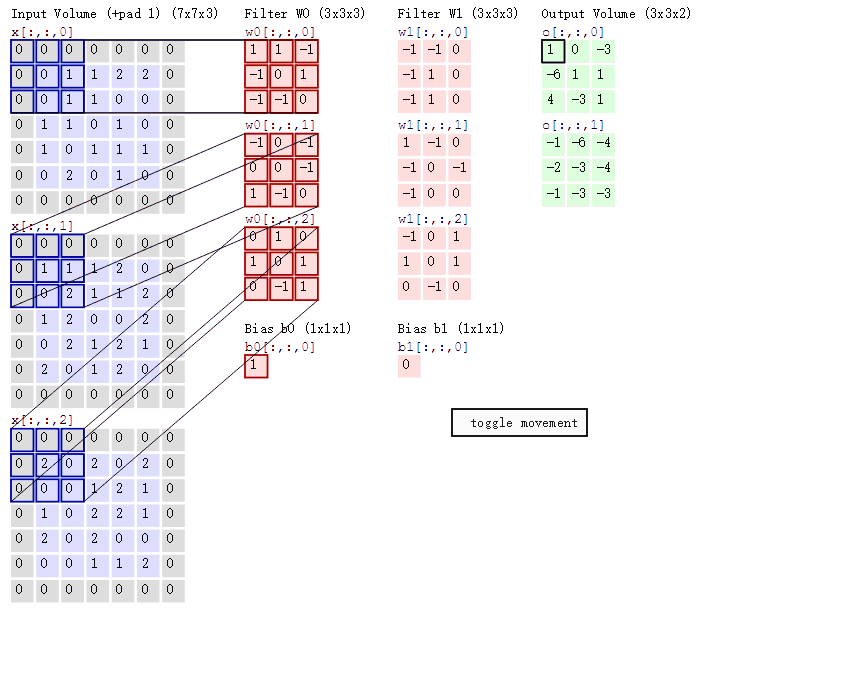
在下图对应的计算过程中，输入是一定区域大小(width\*height)的数据，和滤波器filter（带着一组固定权重的神经元）做内积后等到新的二维数据。

具体来说，左边是图像输入，中间部分就是滤波器filter（带着一组固定权重的神经元），不同的滤波器filter会得到不同的输出数据，比如颜色深浅、轮廓。相当于如果想提取图像的不同特征，则用不同的滤波器filter，提取想要的关于图像的特定信息：颜色深浅或轮廓



**3. CNN滤波器**

在CNN中，滤波器filter（带着一组固定权重的神经元）对局部输入数据进行卷积计算。每计算完一个数据窗口内的局部数据后，数据窗口不断平移滑动，直到计算完所有数据



可以看到

两个神经元，即depth=2，意味着有两个滤波器。

数据窗口每次移动两个步长取3\*3的局部数据，即stride=2。

zero-padding=1

然后分别以两个滤波器filter为轴滑动数组进行卷积计算，得到两组不同的结果。通过这种滑动窗口的滤波过程，逐步把图像的各个细节信息提取出来(边缘轮廓、图像深浅)。值得注意的是

[复制代码](javascript:void(0);)

1. 局部感知机制

左边数据在变化，每次滤波器都是针对某一局部的数据窗口进行卷积，这就是所谓的CNN中的局部感知机制。

打个比方，滤波器就像一双眼睛，人类视角有限，一眼望去，只能看到这世界的局部。如果一眼就看到全世界，你会累死，而且一下子接受全世界所有信息，你大脑接收不过来。当然，即便是看局部，针对局部里的信息人类双眼也是有偏重、偏好的。比如看美女，对脸、胸、腿是重点关注，所以这3个输入的权重相对较大

2. 参数(权重)共享机制

数据窗口滑动，导致输入滤波器的数据在变化，但中间滤波器Filter w0的权重(即每个神经元连接数据窗口的权重)是固定不变的，这个权重不变即所谓的CNN中的参数(权重)共享机制。

再打个比方，某人环游全世界，所看到的信息在变，但采集信息的双眼不变。一个人对景物的认知在一定时间段内是保持不变的，但是需要注意的是，这些权重也不是永远不变的，随着训练的进行，权重会根据激活函数的判断结果不断调整网络中的权重(这就是所谓的BP反向传播算法)

[复制代码](javascript:void(0);)

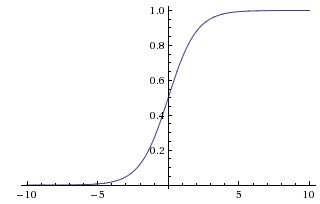
**4. CNN激励层**

常用的非线性激活函数有sigmoid、tanh、relu等等，前两者sigmoid/tanh比较常见于全连接层，后者relu常见于卷积层

***激活函数sigmoid***

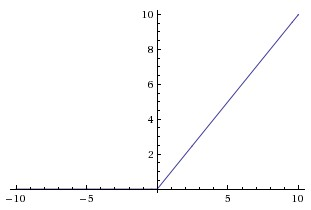
http://images2015.cnblogs.com/blog/532548/201703/532548-20170307110302969-1510553026.png

其中z是一个线性组合，比如z可以等于：b + http://img.blog.csdn.net/20160720151620525\*http://img.blog.csdn.net/20160720151554838 + http://img.blog.csdn.net/20160720151633098\*http://img.blog.csdn.net/20160720151607869



横轴表示定义域z，纵轴表示值域g(z)。sigmoid函数的功能是相当于把一个实数压缩至0到1之间。当z是非常大的正数时，g(z)会趋近于1，而z是非常大的负数时，则g(z)会趋近于0  
这样一来便可以把激活函数看作一种“分类的概率”，比如激活函数的输出为0.9的话便可以解释为90%的概率为正样本

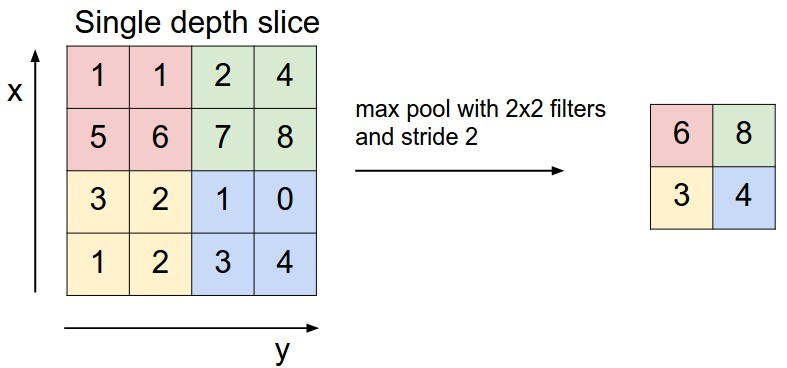
***ReLU激励层***



ReLU的优点是收敛快，求梯度简单

**5. CNN池化层**

池化，简言之，即取区域平均或最大



接下来拿一个真实的CNN网络来解释CNN的构造原理

1. Input layer of NxN pixels (N=32).

2. Convolutional layer (64 filter maps of size 11x11).

3. Max-pooling layer.

4. Densely-connected layer (4096 neurons)

5. Output layer. 9 neurons.

输入图像是一个32\*32的图像集，下面分别解释数据在各层的维度变化

1. input layer: 32x32 neurons

2. convolutional layer(64 filters, size 11x11): (32−11+1)∗(32−11+1) = 22∗22 = 484 for each feature map. As a result, the total output of the convolutional layer is 22∗22∗64 = 30976.

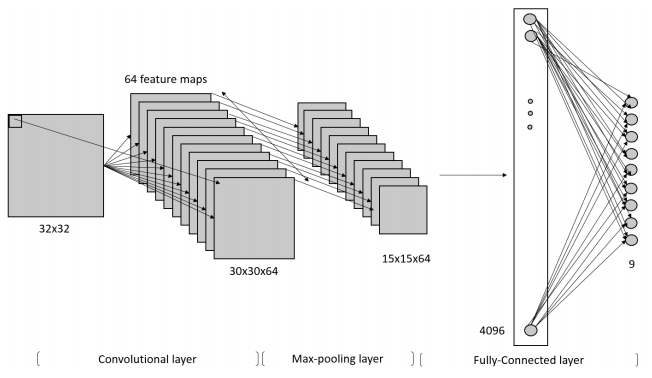
3. pooling layer(2x2 regions): reduced to 11∗11∗64 = 7744.

4. fully-connected layer: 4096 neurons

5. output layer

The number of learnable parameters P of this network is:

P = 1024∗(11∗11∗64)+64+(11∗11∗64)∗4096+4096+4096∗9+9 = 39690313



我们注意看你第二层的CNN层，它实际上可以理解为我们对同一幅图，根据不同的观察重点(滤波窗口移动)得到的不同细节视角的图像

***0x1: Convolution1D层***

一维卷积层，用以在一维输入信号上进行邻域滤波。当使用该层作为首层时，需要提供关键字参数input\_dim或input\_shape。例如input\_dim=128长为128的向量序列输入，而input\_shape=(10,128)代表一个长为10的128向量序列(对于byte词频的代码段特征向量来说就是input\_shape=(15000, 256))

[复制代码](javascript:void(0);)

keras.layers.convolutional.Convolution1D(

nb\_filter,

filter\_length,

init='uniform',

activation='linear',

weights=None,

border\_mode='valid',

subsample\_length=1,

W\_regularizer=None,

b\_regularizer=None,

activity\_regularizer=None,

W\_constraint=None,

b\_constraint=None,

bias=True,

input\_dim=None,

input\_length=None

)

1. nb\_filter：卷积核的数目(即输出的维度)(我们可以利用filter来减少CNN输入层的维度，降低计算量)

2. filter\_length：卷积核的空域或时域长度

3. init：初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的Theano函数。该参数仅在不传递weights参数时有意义。

4. activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）

5. weights：权值，为numpy array的list。该list应含有一个形如（input\_dim,output\_dim）的权重矩阵和一个形如(output\_dim,)的偏置向量。

6. border\_mode：边界模式，为“valid”, “same” 或“full”，full需要以theano为后端

7. subsample\_length：输出对输入的下采样因子

8. W\_regularizer：施加在权重上的正则项，为WeightRegularizer对象

9. b\_regularizer：施加在偏置向量上的正则项，为WeightRegularizer对象

10. activity\_regularizer：施加在输出上的正则项，为ActivityRegularizer对象

11. W\_constraints：施加在权重上的约束项，为Constraints对象

12. b\_constraints：施加在偏置上的约束项，为Constraints对象

13. bias：布尔值，是否包含偏置向量（即层对输入做线性变换还是仿射变换）

14. input\_dim：整数，输入数据的维度。当该层作为网络的第一层时，必须指定该参数或input\_shape参数。

15. input\_length：当输入序列的长度固定时，该参数为输入序列的长度。当需要在该层后连接Flatten层，然后又要连接Dense层时，需要指定该参数，否则全连接的输出无法计算出来

[复制代码](javascript:void(0);)

example

[复制代码](javascript:void(0);)

# apply a convolution 1d of length 3 to a sequence with 10 timesteps,

# with 64 output filters

model = Sequential()

model.add(Convolution1D(64, 3, border\_mode='same', input\_shape=(10, 32)))

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

# add a new conv1d on top

model.add(Convolution1D(32, 3, border\_mode='same'))

# now model.output\_shape == (None, 10, 32)

[复制代码](javascript:void(0);)

可以将Convolution1D看作Convolution2D的快捷版，对例子中（10，32）的信号进行1D卷积相当于对其进行卷积核为（filter\_length, 32）的2D卷积

***0x2: AtrousConvolution1D层***

AtrousConvolution1D层用于对1D信号进行滤波，是膨胀/带孔洞的卷积。当使用该层作为首层时，需要提供关键字参数input\_dim或input\_shape。例如input\_dim=128长为128的向量序列输入，而input\_shape=(10,128)代表一个长为10的128向量序列.

[复制代码](javascript:void(0);)

keras.layers.convolutional.AtrousConvolution1D(

nb\_filter,

filter\_length,

init='uniform',

activation='linear',

weights=None,

border\_mode='valid',

subsample\_length=1,

atrous\_rate=1,

W\_regularizer=None,

b\_regularizer=None,

activity\_regularizer=None,

W\_constraint=None,

b\_constraint=None,

bias=True

)

nb\_filter：卷积核的数目（即输出的维度）

filter\_length：卷积核的空域或时域长度

init：初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的Theano函数。该参数仅在不传递weights参数时有意义。

activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）

weights：权值，为numpy array的list。该list应含有一个形如（input\_dim,output\_dim）的权重矩阵和一个形如(output\_dim,)的偏置向量。

border\_mode：边界模式，为“valid”，“same”或“full”，full需要以theano为后端

subsample\_length：输出对输入的下采样因子

atrous\_rate:卷积核膨胀的系数，在其他地方也被称为'filter\_dilation'

W\_regularizer：施加在权重上的正则项，为WeightRegularizer对象

b\_regularizer：施加在偏置向量上的正则项，为WeightRegularizer对象

activity\_regularizer：施加在输出上的正则项，为ActivityRegularizer对象

W\_constraints：施加在权重上的约束项，为Constraints对象

b\_constraints：施加在偏置上的约束项，为Constraints对象

bias：布尔值，是否包含偏置向量（即层对输入做线性变换还是仿射变换）

input\_dim：整数，输入数据的维度。当该层作为网络的第一层时，必须指定该参数或input\_shape参数。

input\_length：当输入序列的长度固定时，该参数为输入序列的长度。当需要在该层后连接Flatten层，然后又要连接Dense层时，需要指定该参数，否则全连接的输出无法计算出来。

[复制代码](javascript:void(0);)

example

[复制代码](javascript:void(0);)

# apply an atrous convolution 1d with atrous rate 2 of length 3 to a sequence with 10 timesteps,

# with 64 output filters

model = Sequential()

model.add(AtrousConvolution1D(64, 3, atrous\_rate=2, border\_mode='same', input\_shape=(10, 32)))

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

# add a new atrous conv1d on top

model.add(AtrousConvolution1D(32, 3, atrous\_rate=2, border\_mode='same'))

# now model.output\_shape == (None, 10, 32)

[复制代码](javascript:void(0);)

***0x3: Convolution2D层***

二维卷积层对二维输入进行滑动窗卷积，当使用该层作为第一层时，应提供input\_shape参数。例如input\_shape = (3,128,128)代表128\*128的彩色RGB图像

[复制代码](javascript:void(0);)

keras.layers.convolutional.Convolution2D(

nb\_filter,

nb\_row,

nb\_col,

init='glorot\_uniform',

activation='linear',

weights=None,

border\_mode='valid',

subsample=(1, 1),

dim\_ordering='th',

W\_regularizer=None,

b\_regularizer=None,

activity\_regularizer=None,

W\_constraint=None,

b\_constraint=None,

bias=True

)

nb\_filter：卷积核的数目

nb\_row：卷积核的行数

nb\_col：卷积核的列数

init：初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的Theano函数。该参数仅在不传递weights参数时有意义。

activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）

weights：权值，为numpy array的list。该list应含有一个形如（input\_dim,output\_dim）的权重矩阵和一个形如(output\_dim,)的偏置向量。

border\_mode：边界模式，为“valid”，“same”或“full”，full需要以theano为后端

subsample：长为2的tuple，输出对输入的下采样因子，更普遍的称呼是“strides”

W\_regularizer：施加在权重上的正则项，为WeightRegularizer对象

b\_regularizer：施加在偏置向量上的正则项，为WeightRegularizer对象

activity\_regularizer：施加在输出上的正则项，为ActivityRegularizer对象

W\_constraints：施加在权重上的约束项，为Constraints对象

b\_constraints：施加在偏置上的约束项，为Constraints对象

dim\_ordering：‘th’或‘tf’。‘th’模式中通道维（如彩色图像的3通道）位于第1个位置（维度从0开始算），而在‘tf’模式中，通道维位于第3个位置。例如128\*128的三通道彩色图片，在‘th’模式中input\_shape应写为（3，128，128），而在‘tf’模式中应写为（128，128，3），注意这里3出现在第0个位置，因为input\_shape不包含样本数的维度，在其内部实现中，实际上是（None，3，128，128）和（None，128，128，3）。默认是image\_dim\_ordering指定的模式，可在~/.keras/keras.json中查看，若没有设置过则为'tf'。

bias：布尔值，是否包含偏置向量（即层对输入做线性变换还是仿射变换）

[复制代码](javascript:void(0);)

example

[复制代码](javascript:void(0);)

# apply a 3x3 convolution with 64 output filters on a 256x256 image:

model = Sequential()

model.add(Convolution2D(64, 3, 3, border\_mode='same', input\_shape=(3, 256, 256)))

# now model.output\_shape == (None, 64, 256, 256)

# add a 3x3 convolution on top, with 32 output filters:

model.add(Convolution2D(32, 3, 3, border\_mode='same'))

# now model.output\_shape == (None, 32, 256, 256)

[复制代码](javascript:void(0);)

***0x3: AtrousConvolution2D层***

该层对二维输入进行Atrous卷积，也即膨胀卷积或带孔洞的卷积。当使用该层作为第一层时，应提供input\_shape参数。例如input\_shape = (3,128,128)代表128\*128的彩色RGB图像

***Relevant Link:***

https://keras-cn.readthedocs.io/en/latest/layers/convolutional\_layer/

http://baike.baidu.com/item/%E6%BB%A4%E6%B3%A2

http://blog.csdn.net/v\_july\_v/article/details/51812459

http://cs231n.github.io/convolutional-networks/#overview

http://blog.csdn.net/stdcoutzyx/article/details/41596663

**7. 池化层**

***0x1: MaxPooling1D层***

对时域1D信号进行最大值池化

[复制代码](javascript:void(0);)

keras.layers.convolutional.MaxPooling1D(

pool\_length=2,

stride=None,

border\_mode='valid'

)

pool\_length：下采样因子，如取2则将输入下采样到一半长度

stride：整数或None，步长值

border\_mode：‘valid’或者‘same’

[复制代码](javascript:void(0);)

***0x2: MaxPooling2D层***

为空域信号施加最大值池化

[复制代码](javascript:void(0);)

keras.layers.convolutional.MaxPooling2D(

pool\_size=(2, 2),

strides=None,

border\_mode='valid', dim\_ordering='th'

)

1. pool\_size：长为2的整数tuple，代表在两个方向（竖直，水平）上的下采样因子，如取（2，2）将使图片在两个维度上均变为原长的一半

2. strides：长为2的整数tuple，或者None，步长值。

3. border\_mode：‘valid’或者‘same’

4. dim\_ordering：‘th’或‘tf’。‘th’模式中通道维（如彩色图像的3通道）位于第1个位置（维度从0开始算），而在‘tf’模式中，通道维位于第3个位置。例如128\*128的三通道彩色图片，在‘th’模式中input\_shape应写为（3，128，128），而在‘tf’模式中应写为（128，128，3），注意这里3出现在第0个位置，因为input\_shape不包含样本数的维度，在其内部实现中，实际上是（None，3，128，128）和（None，128，128，3）。默认是image\_dim\_ordering指定的模式，可在~/.keras/keras.json中查看，若没有设置过则为'tf'

[复制代码](javascript:void(0);)

***0x3: AveragePooling1D层***

对时域1D信号进行平均值池化

[复制代码](javascript:void(0);)

keras.layers.convolutional.AveragePooling1D(

pool\_length=2,

stride=None,

border\_mode='valid'

)

1. pool\_length：下采样因子，如取2则将输入下采样到一半长度

2. stride：整数或None，步长值

3. border\_mode：‘valid’或者‘same’

注意，目前‘same’模式只能在TensorFlow作为后端时使用

[复制代码](javascript:void(0);)

***0x4: GlobalMaxPooling1D层***

对于时间信号的全局最大池化

keras.layers.pooling.GlobalMaxPooling1D()

***Relevant Link:***

https://keras-cn.readthedocs.io/en/latest/layers/pooling\_layer/

**8. 递归层Recurrent**

***0x1: Recurrent层***

这是递归层的抽象类，请不要在模型中直接应用该层（因为它是抽象类，无法实例化任何对象）。请使用它的子类LSTM或SimpleRNN。  
所有的递归层（LSTM,GRU,SimpleRNN）都服从本层的性质，并接受本层指定的所有关键字参数

[复制代码](javascript:void(0);)

keras.layers.recurrent.Recurrent(

weights=None,

return\_sequences=False,

go\_backwards=False,

stateful=False,

unroll=False,

consume\_less='cpu',

input\_dim=None,

input\_length=None

)

1. weights：numpy array的list，用以初始化权重。该list形如[(input\_dim, output\_dim),(output\_dim, output\_dim),(output\_dim,)]

2. return\_sequences：布尔值，默认False，控制返回类型。若为True则返回整个序列，否则仅返回输出序列的最后一个输出

3. go\_backwards：布尔值，默认为False，若为True，则逆向处理输入序列

4. stateful：布尔值，默认为False，若为True，则一个batch中下标为i的样本的最终状态将会用作下一个batch同样下标的样本的初始状态。

5. unroll：布尔值，默认为False，若为True，则递归层将被展开，否则就使用符号化的循环。当使用TensorFlow为后端时，递归网络本来就是展开的，因此该层不做任何事情。层展开会占用更多的内存，但会加速RNN的运算。层展开只适用于短序列。

6. consume\_less：‘cpu’或‘mem’之一。若设为‘cpu’，则RNN将使用较少、较大的矩阵乘法来实现，从而在CPU上会运行更快，但会更消耗内存。如果设为‘mem’，则RNN将会较多的小矩阵乘法来实现，从而在GPU并行计算时会运行更快（但在CPU上慢），并占用较少内存。

7. input\_dim：输入维度，当使用该层为模型首层时，应指定该值（或等价的指定input\_shape)

8. input\_length：当输入序列的长度固定时，该参数为输入序列的长度。当需要在该层后连接Flatten层，然后又要连接Dense层时，需要指定该参数，否则全连接的输出无法计算出来。注意，如果递归层不是网络的第一层，你需要在网络的第一层中指定序列的长度，如通过input\_shape指定。

[复制代码](javascript:void(0);)

***0x2: SimpleRNN层***

全连接RNN网络，RNN的输出会被回馈到输入

[复制代码](javascript:void(0);)

keras.layers.recurrent.SimpleRNN(

output\_dim,

init='glorot\_uniform',

inner\_init='orthogonal',

activation='tanh',

W\_regularizer=None,

U\_regularizer=None,

b\_regularizer=None,

dropout\_W=0.0,

dropout\_U=0.0

)

output\_dim：内部投影和输出的维度

init：初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的Theano函数。

inner\_init：内部单元的初始化方法

activation：激活函数，为预定义的激活函数名（参考激活函数）

W\_regularizer：施加在权重上的正则项，为WeightRegularizer对象

U\_regularizer：施加在递归权重上的正则项，为WeightRegularizer对象

b\_regularizer：施加在偏置向量上的正则项，为WeightRegularizer对象

dropout\_W：0~1之间的浮点数，控制输入单元到输入门的连接断开比例

dropout\_U：0~1之间的浮点数，控制输入单元到递归连接的断开比例

[复制代码](javascript:void(0);)

***0x3: GRU层***

门限递归单元

[复制代码](javascript:void(0);)

keras.layers.recurrent.GRU(

output\_dim,

init='glorot\_uniform',

inner\_init='orthogonal',

activation='tanh',

inner\_activation='hard\_sigmoid',

W\_regularizer=None,

U\_regularizer=None,

b\_regularizer=None,

dropout\_W=0.0,

dropout\_U=0.0

)

output\_dim：内部投影和输出的维度

init：初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的Theano函数。

inner\_init：内部单元的初始化方法

activation：激活函数，为预定义的激活函数名（参考激活函数）

inner\_activation：内部单元激活函数

W\_regularizer：施加在权重上的正则项，为WeightRegularizer对象

U\_regularizer：施加在递归权重上的正则项，为WeightRegularizer对象

b\_regularizer：施加在偏置向量上的正则项，为WeightRegularizer对象

dropout\_W：0~1之间的浮点数，控制输入单元到输入门的连接断开比例

dropout\_U：0~1之间的浮点数，控制输入单元到递归连接的断开比例

[复制代码](javascript:void(0);)

***0x4: LSTM层***

Keras长短期记忆模型

[复制代码](javascript:void(0);)

keras.layers.recurrent.LSTM(

output\_dim,

init='glorot\_uniform',

inner\_init='orthogonal',

forget\_bias\_init='one',

activation='tanh',

inner\_activation='hard\_sigmoid',

W\_regularizer=None,

U\_regularizer=None,

b\_regularizer=None,

dropout\_W=0.0,

dropout\_U=0.0

)

output\_dim：内部投影和输出的维度

init：初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的Theano函数。

inner\_init：内部单元的初始化方法

forget\_bias\_init：遗忘门偏置的初始化函数，Jozefowicz et al.建议初始化为全1元素

activation：激活函数，为预定义的激活函数名（参考激活函数）

inner\_activation：内部单元激活函数

W\_regularizer：施加在权重上的正则项，为WeightRegularizer对象

U\_regularizer：施加在递归权重上的正则项，为WeightRegularizer对象

b\_regularizer：施加在偏置向量上的正则项，为WeightRegularizer对象

dropout\_W：0~1之间的浮点数，控制输入单元到输入门的连接断开比例

dropout\_U：0~1之间的浮点数，控制输入单元到递归连接的断开比例

[复制代码](javascript:void(0);)

***Relevant Link:***

https://keras-cn.readthedocs.io/en/latest/layers/recurrent\_layer/

**9. 嵌入层 Embedding**

***0x1: Embedding层***

嵌入层将正整数（下标）转换为具有固定大小的向量，如[[4],[20]]->[[0.25,0.1],[0.6,-0.2]]。是一种数字化->向量化的编码方式，使用Embedding需要输入的特征向量具备空间关联性  
Embedding层只能作为模型的第一层

[复制代码](javascript:void(0);)

keras.layers.embeddings.Embedding(

input\_dim,

output\_dim,

init='uniform',

input\_length=None,

W\_regularizer=None,

activity\_regularizer=None,

W\_constraint=None,

mask\_zero=False,

weights=None,

dropout=0.0

)

input\_dim：大或等于0的整数，字典长度，即输入数据最大下标+1

output\_dim：大于0的整数，代表全连接嵌入的维度

init：初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的Theano函数。该参数仅在不传递weights参数时有意义。

weights：权值，为numpy array的list。该list应仅含有一个如（input\_dim,output\_dim）的权重矩阵

W\_regularizer：施加在权重上的正则项，为WeightRegularizer对象

W\_constraints：施加在权重上的约束项，为Constraints对象

mask\_zero：布尔值，确定是否将输入中的‘0’看作是应该被忽略的‘填充’（padding）值，该参数在使用递归层处理变长输入时有用。设置为True的话，模型中后续的层必须都支持masking，否则会抛出异常

input\_length：当输入序列的长度固定时，该值为其长度。如果要在该层后接Flatten层，然后接Dense层，则必须指定该参数，否则Dense层的输出维度无法自动推断。

dropout：0~1的浮点数，代表要断开的嵌入比例

[复制代码](javascript:void(0);)

***Relevant Link:***

https://keras-cn.readthedocs.io/en/latest/layers/embedding\_layer/

***Copyright (c) 2017 LittleHann All rights reserved***