



# Keras: 基于 Python 的深度学习库



你恰好发现了 Keras。

Keras 是一个用 Python 编写的高级神经网络 API，它能够以 [TensorFlow](#), [CNTK](#) 或者 [Theano](#) 作为后端运行。Keras 的开发重点是支持快速的实验。能够以最小的时延把你的想法转换为实验结果，是做好研究的关键。

如果你在以下情况下需要深度学习库，请使用 Keras:

- 允许简单而快速的原型设计（由于用户友好，高度模块化，可扩展性）。
- 同时支持卷积神经网络和循环神经网络，以及两者的组合。
- 在 CPU 和 GPU 上无缝运行。

查看文档，请访问 [Keras.io](#)。备份网址：[Keras-zh](#)。

Keras 兼容的 Python 版本: **Python 2.7-3.6**。

多后端 Keras 和 tf.keras:

目前，我们推荐使用 [TensorFlow](#) 后端的 Keras 用户切换至 [TensorFlow 2.0](#) 的 `tf.keras`。

`tf.keras` 具有更好的维护，并且更好地集成了 [TensorFlow](#) 功能（eager 执行，分布式支持及其他）。

Keras 2.2.5 是最后一个实现 2.2.\* API 的 Keras 版本。它是最后一个仅支持 [TensorFlow 1](#)（以及 [Theano](#) 和 [CNTK](#)）的版本。

Keras 的当前版本是 2.3.0，它对 API 做了重大的调整，并且添加了 [TensorFlow 2.0](#) 的支持。

2.3.0 将会是最后个多后端 Keras 主版本。多后端 Keras 已被 `tf.keras` 取代。

多后端 Keras 中存在的错误修复仅会持续到 2020 年 4 月（作为次要版本的一部分）。

关于 Keras 未来的更多信息，详见 [the Keras meeting notes](#)。

## 指导原则

- **用户友好。** Keras 是为人类而不是为机器设计的 API。它把用户体验放在首要和中心位置。Keras 遵循减少认知困难的最佳实践：它提供一致且简单的 API，将常见用例所需的用户操作数量降至最低，并且在用户错误时提供清晰和可操作的反馈。
- **模块化。** 模型被理解为由独立的、完全可配置的模块构成的序列或图。这些模块可以以尽可能少的限制组装在一起。特别是神经网络层、损失函数、优化器、初始化方法、激活函数、正则化方法，它们都是可以结合起来构建新模型的模块。
- **易扩展性。** 新的模块很容易添加的（作为新的类和函数），现有的模块已经提供了充足的示例。由于能够轻松地创建可以提高表现力的新模块，Keras 更加适合高级研究。
- **基于 Python 实现。** Keras 没有特定格式的单独配置文件。模型定义在 Python 代码中，这些代码紧凑，易于调试，并且易于扩展。

## 快速开始：30 秒上手 Keras

Keras 的核心数据结构是 **model**，一种组织网络层的方式。最简单的模型是 **Sequential 顺序模型**，它由多个网络层线性堆叠。对于更复杂的结构，你应该使用 **Keras 函数式 API**，它允许构建任意的神经网络图。

Sequential 模型如下所示：

```
from keras.models import Sequential  
  
model = Sequential()
```

可以简单地使用 `.add()` 来堆叠模型：

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

在完成了模型的构建后，可以使用 `.compile()` 来配置学习过程：

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

如果需要，你还可以进一步地配置你的优化器。Keras 的核心原则是使事情变得相当简单，同时又允许用户在需要的时候能够进行完全的控制（终极的控制是源代码的易扩展性）。

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

现在，你可以批量地在训练数据上进行迭代了：

```
# x_train 和 y_train 是 Numpy 数组 -- 就像在 Scikit-Learn API 中一样。
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

或者，你可以手动地将批次的数据提供给模型：

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

只需一行代码就能评估模型性能：

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

或者对新的数据生成预测：

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

构建一个问答系统，一个图像分类模型，一个神经图灵机，或者其他任何模型，就这么的快。深度学习背后的思想很简单，那么它们的实现又何必要那么痛苦呢？

有关 Keras 更深入的教程，请查看：

- [开始使用 Sequential 模型](#)
- [开始使用函数式 API](#)

在代码仓库的 [examples](#) 目录中，你会找到更多高级模型：基于记忆网络的问答系统、基于栈式 LSTM 的文本生成等等。

## 安装指引

在安装 Keras 之前，请安装以下后端引擎之一：TensorFlow, Theano 或者 CNTK。我们推荐 TensorFlow 后端。

- [TensorFlow 安装指引。](#)
- [Theano 安装指引。](#)

- CNTK 安装指引。

你也可以考虑安装以下\*\*可选依赖\*\*：

- cuDNN (如果你计划在 GPU 上运行 Keras，建议安装)。
- HDF5 和 h5py (如果你需要将 Keras 模型保存到磁盘，则需要这些)。
- graphviz 和 pydot (用于绘制模型图的可视化工具)。

然后你就可以安装 Keras 本身了。有两种方法安装 Keras：

- **使用 PyPI 安装 Keras (推荐) :**

注意：这些安装步骤假定你在 Linux 或 Mac 环境中。如果你使用的是 Windows，则需要删除 sudo 才能运行以下命令。

```
sudo pip install keras
```

如果你使用 virtualenv 虚拟环境，你可以避免使用 sudo：

```
pip install keras
```

- **或者：使用 GitHub 源码安装 Keras：**

首先，使用 git 来克隆 Keras：

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

然后， cd 到 Keras 目录并且运行安装命令：

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

## 配置你的 Keras 后端

默认情况下，Keras 将使用 TensorFlow 作为其张量操作库。请[跟随这些指引](#)来配置其他 Keras 后端。

## 技术支持

你可以提出问题并参与开发讨论：

- [Keras Google group](#)。
- [Keras Slack channel](#)。使用 [这个链接](#) 向该频道请求邀请函。
- 或者加入 Keras 深度学习交流群，协助文档的翻译工作，群号为 951623081。

你也可以在 [GitHub issues](#) 中发布\*\*漏洞报告和新功能请求\*\*（仅限于此）。注意请先阅读[规范文档](#)。

## 为什么取名为 Keras?

Keras (*κέρας*) 在希腊语中意为 号角。它来自古希腊和拉丁文学中的一个文学形象，首先出现于《奥德赛》中，梦神 (*Oneiroi*, singular *Oneiros*) 从这两类人中分离出来：那些用虚幻的景象欺骗人类，通过象牙之门抵达地球之人，以及那些宣告未来即将到来，通过号角之门抵达之人。它类似于文字寓意，*κέρας* (号角) / *κραίνω* (履行)，以及 *ἐλέφας* (象牙) / *ἐλεφαίρομαι* (欺骗)。

Keras 最初是作为 ONEIROS 项目（开放式神经电子智能机器人操作系统）研究工作的一部分而开发的。

“*Oneiroi* 超出了我们的理解 - 谁能确定它们讲述了什么故事？并不是所有人都能找到。那里有两扇门，就是通往短暂的 *Oneiroi* 的通道；一个是用号角制造的，一个是用象牙制造的。穿过尖锐的象牙的 *Oneiroi* 是诡计多端的，他们带有一些不会实现的信息；那些穿过抛光的喇叭出来的人背后具有真理，对于看到他们的人来说是完成的。” Homer, Odyssey 19. 562 ff (Shewring translation).



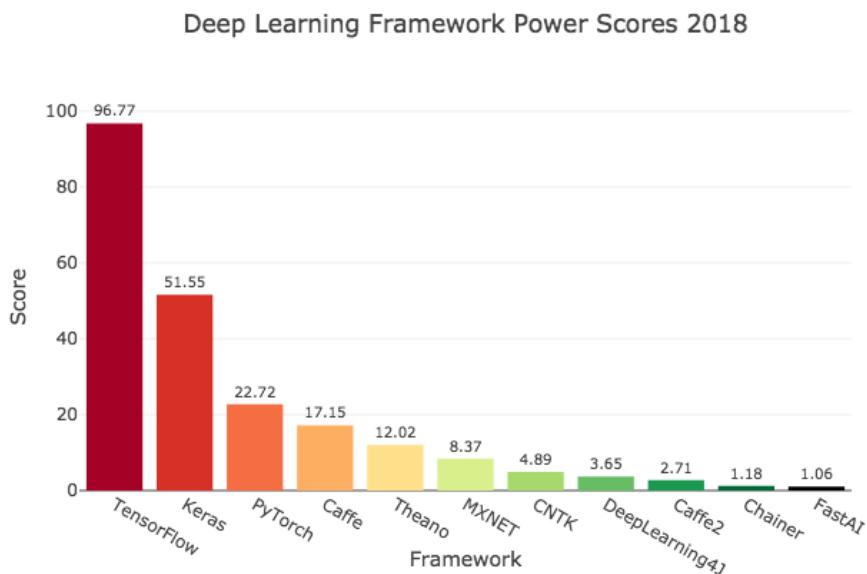
# 为什么选择 Keras?

在如今无数深度学习框架中，为什么要使用 Keras 而非其他？以下是 Keras 与现有替代品的一些比较。

## Keras 优先考虑开发人员的经验

- Keras 是为人类而非机器设计的 API。[Keras 遵循减少认知困难的最佳实践](#): 它提供一致且简单的 API，它将常见用例所需的用户操作数量降至最低，并且在用户错误时提供清晰和可操作的反馈。
- 这使 Keras 易于学习和使用。作为 Keras 用户，你的工作效率更高，能够比竞争对手更快地尝试更多创意，从而[帮助你赢得机器学习竞赛](#)。
- 这种易用性并不以降低灵活性为代价：因为 Keras 与底层深度学习语言（特别是 TensorFlow）集成在一起，所以它可以让你实现任何你可以用基础语言编写的东西。特别是，`tf.keras` 作为 Keras API 可以与 TensorFlow 工作流无缝集成。

## Keras 被工业界和学术界广泛采用



Deep learning 框架排名，由 Jeff Hale 基于 7 个分类的 11 个数据源计算得出

截至 2018 年中期，Keras 拥有超过 250,000 名个人用户。与其他任何深度学习框架相比，Keras 在行业和研究领域的应用率更高（除 TensorFlow 之外，且 Keras API 是 TensorFlow 的官方前端，通过 `tf.keras` 模块使用）。

你已经不断与使用 Keras 构建的功能进行交互 - 它在 Netflix, Uber, Yelp, Instacart, Zocdoc, Square 等众多网站上被使用。它尤其受以深度学习作为产品核心的创业公司的欢迎。

Keras 也是深度学习研究人员的最爱，在上传到预印本服务器 [arXiv.org](#) 的科学论文中被提及的次数位居第二。Keras 还被大型科学组织的研究人员采用，特别是 CERN 和 NASA。

## Keras 可以轻松将模型转化为产品

与任何其他深度学习框架相比，你的 Keras 模型可以轻松地部署在更广泛的平台上：

- 在 iOS 上，通过 [Apple's CoreML](#)（苹果为 Keras 提供官方支持）。这里有一个[教程](#)。
- 在 Android 上，通过 TensorFlow Android runtime。例如 [Not Hotdog app](#)。
- 在浏览器中，通过 GPU 加速的 JavaScript 运行时，例如 [Keras.js](#) 和 [WebDNN](#)。
- 在 Google Cloud 上，通过 [TensorFlow-Serving](#)。
- 在 Python webapp 后端中（比如 Flask app）。
- 在 JVM 上，通过 [SkyMind](#) 提供的 DL4J 模型导入。
- 在 Raspberry Pi 树莓派上。

## Keras 支持多个后端引擎，不会将你锁定到一个生态系统中

你的 Keras 模型可以基于不同的[深度学习后端](#)开发。重要的是，任何仅利用内置层构建的 Keras 模型，都可以在所有这些后端中移植：你可以用一种后端训练模型，再将它载入另一种后端中（例如为了发布的需要）。支持的后端有：

- 谷歌的 TensorFlow 后端
- 微软的 CNTK 后端
- Theano 后端

亚马逊也有一个[使用 MXNet 作为后端的 Keras 分支](#)。

如此一来，你的 Keras 模型可以在 CPU 之外的不同硬件平台上训练：

- NVIDIA GPUs
- Google TPUs，通过 TensorFlow 后端和 Google Cloud

- OpenCL 支持的 GPUs，比如 AMD，通过 [PlaidML Keras 后端](#)

## Keras 拥有强大的多 GPU 和分布式训练支持

- Keras [内置对多 GPU 数据并行的支持。](#)
- 优步的 [Horovod](#) 对 Keras 模型拥有一流的支持。
- Keras 模型[可以被转换为 TensorFlow Estimators](#)并在 Google Cloud 的 GPU 集群上训练。
- Keras 可以在 Spark（通过 CERN 的 [Dist-Keras](#)）和 [Elephas](#) 上运行。

## Keras 的发展得到深度学习生态系统中的关键公司的支持

Keras 的开发主要由谷歌支持，Keras API 以 `tf.keras` 的形式包装在 TensorFlow 中。此外，微软维护着 Keras 的 CNTK 后端。亚马逊 AWS 正在开发 MXNet 支持。其他提供支持的公司包括 NVIDIA、优步、苹果（通过 CoreML）等。



# 开始使用 Keras Sequential 顺序模型

顺序模型是多个网络层的线性堆叠。

你可以通过将网络层实例的列表传递给 `Sequential` 的构造器，来创建一个 `Sequential` 模型：

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

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

也可以简单地使用 `.add()` 方法将各层添加到模型中：

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

## 指定输入数据的尺寸

模型需要知道它所期望的输入的尺寸。出于这个原因，顺序模型中的第一层（且只有第一层，因为下面的层可以自动地推断尺寸）需要接收关于其输入尺寸的信息。有几种方法来做到这一点：

- 传递一个 `input_shape` 参数给第一层。它是一个表示尺寸的元组（一个由整数或 `None` 组成的元组，其中 `None` 表示可能为任何正整数）。在 `input_shape` 中不包含数据的 batch 大小。
- 某些 2D 层，例如 `Dense`，支持通过参数 `input_dim` 指定输入尺寸，某些 3D 时序层支持 `input_dim` 和 `input_length` 参数。
- 如果你需要为你的输入指定一个固定的 batch 大小（这对 stateful RNNs 很有用），你可以传递一个 `batch_size` 参数给一个层。如果你同时将 `batch_size=32` 和 `input_shape=(6, 8)` 传递给一个层，那么每一批输入的尺寸就为 `(32, 6, 8)`。

因此，下面的代码片段是等价的：

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

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

## 模型编译

在训练模型之前，您需要配置学习过程，这是通过 `compile` 方法完成的。它接收三个参数：

- 优化器 `optimizer`。它可以是现有优化器的字符串标识符，如 `rmsprop` 或 `adagrad`，也可以是 `Optimizer` 类的实例。详见：[optimizers](#)。
- 损失函数 `loss`，模型试图最小化的目标函数。它可以是现有损失函数的字符串标识符，如 `categorical_crossentropy` 或 `mse`，也可以是一个目标函数。详见：[losses](#)。
- 评估标准 `metrics`。对于任何分类问题，你都希望将其设置为 `metrics = ['accuracy']`。评估标准可以是现有的标准的字符串标识符，也可以是自定义的评估标准函数。详见：[metrics](#)。

```
# 多分类问题
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# 二分类问题
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# 均方误差回归问题
model.compile(optimizer='rmsprop',
              loss='mse')

# 自定义评估标准函数
import keras.backend as K

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

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

## 模型训练

Keras 模型在输入数据和标签的 Numpy 矩阵上进行训练。为了训练一个模型，你通常会使用 `fit` 函数。[文档详见此处](#)。

```
# 对于具有 2 个类的单输入模型（二进制分类）：

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

# 生成虚拟数据
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))

# 训练模型，以 32 个样本为一个 batch 进行迭代
model.fit(data, labels, epochs=10, batch_size=32)
```

```
# 对于具有 10 个类的单输入模型（多分类分类）：

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

# 生成虚拟数据
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))

# 将标签转换为分类的 one-hot 编码
one_hot_labels = keras.utils.to_categorical(labels, num_classes=10)

# 训练模型，以 32 个样本为一个 batch 进行迭代
model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

## 示例

这里有几个可以帮助你起步的例子！

在 [examples](#) 目录中，你可以找到真实数据集的示例模型：

- CIFAR10 小图片分类：具有实时数据增强的卷积神经网络 (CNN)
- IMDB 电影评论情感分类：基于词序列的 LSTM
- Reuters 新闻主题分类：多层感知器 (MLP)
- MNIST 手写数字分类：MLP & CNN
- 基于 LSTM 的字符级文本生成

...以及更多。

## 基于多层感知器 (MLP) 的 softmax 多分类：

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

# 生成虚拟数据
import numpy as np
x_train = np.random.random((1000, 20))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)),
num_classes=10)
x_test = np.random.random((100, 20))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)),
num_classes=10)

model = Sequential()
# Dense(64) 是一个具有 64 个隐藏神经元的全连接层。
# 在第一层必须指定所期望的输入数据尺寸：
# 在这里，是一个 20 维的向量。
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

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

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

## 基于多层感知器的二分类：

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

# 生成虚拟数据
x_train = np.random.random((1000, 20))
y_train = np.random.randint(2, size=(1000, 1))
x_test = np.random.random((100, 20))
y_test = np.random.randint(2, size=(100, 1))

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

```

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

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

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

```

类似 VGG 的卷积神经网络：

```

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

# 生成虚拟数据
x_train = np.random.random((100, 100, 100, 3))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)),
                                     num_classes=10)
x_test = np.random.random((20, 100, 100, 3))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(20, 1)),
                                     num_classes=10)

model = Sequential()
# 输入：3 通道 100x100 像素图像 -> (100, 100, 3) 张量。
# 使用 32 个大小为 3x3 的卷积滤波器。
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(100, 100, 3)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

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

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

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

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

```

## 基于 LSTM 的序列分类：

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

max_features = 1024

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

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

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

## 基于 1D 卷积的序列分类：

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

seq_length = 64

model = Sequential()
model.add(Conv1D(64, 3, activation='relu', input_shape=(seq_length, 100)))
model.add(Conv1D(64, 3, activation='relu'))
model.add(MaxPooling1D(3))
model.add(Conv1D(128, 3, activation='relu'))
model.add(Conv1D(128, 3, activation='relu'))
model.add(GlobalAveragePooling1D())
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

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

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

## 基于栈式 LSTM 的序列分类

在这个模型中，我们将 3 个 LSTM 层叠在一起，使模型能够学习更高层次的时间表示。

前两个 LSTM 返回完整的输出序列，但最后一个只返回输出序列的最后一步，从而降低了时间维度（即将输入序列转换成单个向量）。

### stacked LSTM

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

data_dim = 16
timesteps = 8
num_classes = 10

# 期望输入数据尺寸: (batch_size, timesteps, data_dim)
model = Sequential()
model.add(LSTM(32, return_sequences=True,
              input_shape=(timesteps, data_dim))) # 返回维度为 32 的向量序列
model.add(LSTM(32, return_sequences=True)) # 返回维度为 32 的向量序列
model.add(LSTM(32)) # 返回维度为 32 的单个向量
model.add(Dense(10, activation='softmax'))

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

# 生成虚拟训练数据
x_train = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, num_classes))

# 生成虚拟验证数据
x_val = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, num_classes))

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

### “stateful” 渲染的的栈式 LSTM 模型

有状态 (stateful) 的循环神经网络模型中，在一个 batch 的样本处理完成后，其内部状态（记忆）会被记录并作为下一个 batch 的样本的初始状态。这允许处理更长的序列，同时保持计算复杂度的可控性。

你可以在 FAQ 中查找更多关于 stateful RNNs 的信息。

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

data_dim = 16
timesteps = 8
num_classes = 10
```

```
batch_size = 32

# 期望输入数据尺寸: (batch_size, timesteps, data_dim)
# 请注意，我们必须提供完整的 batch_input_shape，因为网络是有状态的。
# 第 k 批数据的第 i 个样本是第 k-1 批数据的第 i 个样本的后续。
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'])

# 生成虚拟训练数据
x_train = np.random.random((batch_size * 10, timesteps, data_dim))
y_train = np.random.random((batch_size * 10, num_classes))

# 生成虚拟验证数据
x_val = np.random.random((batch_size * 3, timesteps, data_dim))
y_val = np.random.random((batch_size * 3, num_classes))

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



# 开始使用 Keras 函数式 API

Keras 函数式 API 是定义复杂模型（如多输出模型、有向无环图或具有共享层的模型）的方法。

这部分文档假设你已经对 `Sequential` 顺序模型比较熟悉。

让我们先从一些简单的示例开始。

## 例一：全连接网络

`Sequential` 模型可能是实现这种网络的一个更好选择，但这个例子能够帮助我们进行一些简单的理解。

- 网络层的实例是可调用的，它以张量为参数，并且返回一个张量
- 输入和输出均为张量，它们都可以用来定义一个模型（`Model`）
- 这样的模型同 Keras 的 `Sequential` 模型一样，都可以被训练

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

# 这部分返回一个张量
inputs = Input(shape=(784,))

# 层的实例是可调用的，它以张量为参数，并且返回一个张量
output_1 = Dense(64, activation='relu')(inputs)
output_2 = Dense(64, activation='relu')(output_1)
predictions = Dense(10, activation='softmax')(output_2)

# 这部分创建了一个包含输入层和三个全连接层的模型
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels) # 开始训练
```

## 所有的模型都可调用，就像网络层一样

利用函数式 API，可以轻易地重用训练好的模型：可以将任何模型看作是一个层，然后通过传递一个张量来调用它。注意，在调用模型时，您不仅重用模型的\*结构\*，还重用了它的权重。

```
x = Input(shape=(784, ))
# 这是可行的，并且返回上面定义的 10-way softmax。
y = model(x)
```

这种方式能允许我们快速创建可以处理\*序列输入\*的模型。只需一行代码，你就将图像分类模型转换为视频分类模型。

```
from keras.layers import TimeDistributed

# 输入张量是 20 个时间步的序列,
# 每一个时间为一个 784 维的向量
input_sequences = Input(shape=(20, 784))

# 这部分将我们之前定义的模型应用于输入序列中的每个时间步。
# 之前定义的模型的输出是一个 10-way softmax,
# 因而下面的层的输出将是维度为 10 的 20 个向量的序列。
processed_sequences = TimeDistributed(model)(input_sequences)
```

## 多输入多输出模型

以下是函数式 API 的一个很好的例子：具有多个输入和输出的模型。函数式 API 使处理大量交织的数据流变得容易。

来考虑下面的模型。我们试图预测 Twitter 上的一条新闻标题有多少转发和点赞数。模型的主要输入将是新闻标题本身，即一系列词语，但是为了增添趣味，我们的模型还添加了其他的辅助输入来接收额外的数据，例如新闻标题的发布时间等。该模型也将通过两个损失函数进行监督学习。较早地在模型中使用主损失函数，是深度学习模型的一个良好正则方法。

模型结构如下图所示：

multi-input-multi-output-graph

让我们用函数式 API 来实现它。

主要输入接收新闻标题本身，即一个整数序列（每个整数编码一个词）。这些整数在 1 到 10,000 之间（10,000 个词的词汇表），且序列长度为 100 个词。

```
from keras.layers import Input, Embedding, LSTM, Dense
from keras.models import Model
import numpy as np
np.random.seed(0) # 设置随机种子，用于复现结果

# 标题输入：接收一个含有 100 个整数的序列，每个整数在 1 到 10000 之间。
# 注意我们可以通过传递一个 "name" 参数来命名任何层。
main_input = Input(shape=(100,), dtype='int32', name='main_input')

# Embedding 层将输入序列编码为一个稠密向量的序列，
```

```
# 每个向量维度为 512。  
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)  
  
# LSTM 层把向量序列转换成单个向量，  
# 它包含整个序列的上下文信息  
lstm_out = LSTM(32)(x)
```

在这里，我们插入辅助损失，使得即使在模型主损失很高的情况下，LSTM 层和 Embedding 层都能被平稳地训练。

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

此时，我们将辅助输入数据与 LSTM 层的输出连接起来，输入到模型中：

```
auxiliary_input = Input(shape=(5,), name='aux_input')  
x = keras.layers.concatenate([lstm_out, auxiliary_input])  
  
# 堆叠多个全连接网络层  
x = Dense(64, activation='relu')(x)  
x = Dense(64, activation='relu')(x)  
x = Dense(64, activation='relu')(x)  
  
# 最后添加主要的逻辑回归层  
main_output = Dense(1, activation='sigmoid', name='main_output')(x)
```

然后定义一个具有两个输入和两个输出的模型：

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

现在编译模型，并给辅助损失分配一个 0.2 的权重。如果要为不同的输出指定不同的 `loss_weights` 或 `loss`，可以使用列表或字典。在这里，我们给 `loss` 参数传递单个损失函数，这个损失将用于所有的输出。

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

我们可以通过传递输入数组和目标数组的列表来训练模型：

```
headline_data = np.round(np.abs(np.random.rand(12, 100) * 100))  
additional_data = np.random.randn(12, 5)  
headline_labels = np.random.randn(12, 1)  
additional_labels = np.random.randn(12, 1)  
model.fit([headline_data, additional_data], [headline_labels,  
additional_labels],  
epochs=50, batch_size=32)
```

由于输入和输出均被命名了（在定义时传递了一个 `name` 参数），我们也可以通过以下方式编译模型：

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

# 然后使用以下方式训练：
model.fit({'main_input': headline_data, 'aux_input': additional_data},
          {'main_output': headline_labels, 'aux_output': additional_labels},
          epochs=50, batch_size=32)
```

若使用此模型做推理，可以

```
model.predict({'main_input': headline_data, 'aux_input': additional_data})
```

或者

```
pred = model.predict([headline_data, additional_data])
```

## 共享网络层

函数式 API 的另一个用途是使用共享网络层的模型。我们来看看共享层。

来考虑推特推文数据集。我们想要建立一个模型来分辨两条推文是否来自同一个人（例如，通过推文的相似性来对用户进行比较）。

实现这个目标的一种方法是建立一个模型，将两条推文编码成两个向量，连接向量，然后添加逻辑回归层；这将输出两条推文来自同一作者的概率。模型将接收一对对正负表示的推特数据。

由于这个问题是对称的，编码第一条推文的机制应该被完全重用来编码第二条推文（权重及其他全部）。这里我们使用一个共享的 LSTM 层来编码推文。

让我们使用函数式 API 来构建它。首先我们将一条推特转换为一个尺寸为 (280, 256) 的矩阵，即每条推特 280 字符，每个字符为 256 维的 one-hot 编码向量（取 256 个常用字符）。

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

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

要在不同的输入上共享同一个层，只需实例化该层一次，然后根据需要传入你想要的输入即可：

```
# 这一层可以输入一个矩阵，并返回一个 64 维的向量
shared_lstm = LSTM(64)
```

```

# 当我们重用相同的图层实例多次，图层的权重也会被重用（它其实就是同一层）
encoded_a = shared_lstm(tweet_a)
encoded_b = shared_lstm(tweet_b)

# 然后再连接两个向量：
merged_vector = keras.layers.concatenate([encoded_a, encoded_b], axis=-1)

# 再在上面添加一个逻辑回归层
predictions = Dense(1, activation='sigmoid')(merged_vector)

# 定义一个连接推特输入和预测的可训练的模型
model = Model(inputs=[tweet_a, tweet_b], outputs=predictions)

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

```

让我们暂停一会，看看如何读取共享层的输出或输出尺寸。

## 层「节点」的概念

每当你在某个输入上调用一个层时，都将创建一个新的张量（层的输出），并且为该层添加一个「节点」，将输入张量连接到输出张量。当多次调用同一个图层时，该图层将拥有多个节点索引（0, 1, 2...）。

在之前版本的 Keras 中，可以通过 `layer.get_output()` 来获得层实例的输出张量，或者通过 `layer.output_shape` 来获取其输出形状。现在你依然可以这么做（除了 `get_output()` 已经被 `output` 属性替代）。但是如果一个层与多个输入连接呢？

只要一个层仅仅连接到一个输入，就不会有困惑，`.output` 会返回层的唯一输出：

```

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

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

assert lstm.output == encoded_a

```

但是如果该层有多个输入，那就会出现问题：

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

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

lstm.output
```

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

好吧，通过下面的方法可以解决：

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

够简单，对吧？

`input_shape` 和 `output_shape` 这两个属性也是如此：只要该层只有一个节点，或者只要所有节点具有相同的输入/输出尺寸，那么「层输出/输入尺寸」的概念就被很好地定义，且将由 `layer.output_shape` / `layer.input_shape` 返回。但是比如说，如果将一个 `Conv2D` 层先应用于尺寸为 `(32, 32, 3)` 的输入，再应用于尺寸为 `(64, 64, 3)` 的输入，那么这个层就会有多个输入/输出尺寸，你将不得不通过指定它们所属节点的索引来获取它们：

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

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

# 到目前为止只有一个输入，以下可行：
assert conv.input_shape == (None, 32, 32, 3)

conved_b = conv(b)
# 现在 `input_shape` 属性不可行，但是这样可以：
assert conv.get_input_shape_at(0) == (None, 32, 32, 3)
assert conv.get_input_shape_at(1) == (None, 64, 64, 3)
```

## 更多的例子

代码示例仍然是起步的最佳方式，所以这里还有更多的例子。

## Inception 模型

有关 Inception 结构的更多信息，请参阅 [Going Deeper with Convolutions](#)。

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

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

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

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

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

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

## 卷积层上的残差连接

有关残差网络 (Residual Network) 的更多信息，请参阅 [Deep Residual Learning for Image Recognition](#)。

```
from keras.layers import Conv2D, Input

# 输入张量为 3 通道 256x256 图像
x = Input(shape=(256, 256, 3))
# 3 输出通道（与输入通道相同）的 3x3 卷积核
y = Conv2D(3, (3, 3), padding='same')(x)
# 返回 x + y
z = keras.layers.add([x, y])
```

## 共享视觉模型

该模型在两个输入上重复使用同一个图像处理模块，以判断两个 MNIST 数字是否为相同的数字。

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

# 首先，定义视觉模型
digit_input = Input(shape=(27, 27, 1))
x = Conv2D(64, (3, 3))(digit_input)
x = Conv2D(64, (3, 3))(x)
x = MaxPooling2D((2, 2))(x)
out = Flatten()(x)

vision_model = Model(digit_input, out)
```

```

# 然后，定义区分数字的模型
digit_a = Input(shape=(27, 27, 1))
digit_b = Input(shape=(27, 27, 1))

# 视觉模型将被共享，包括权重和其他所有
out_a = vision_model(digit_a)
out_b = vision_model(digit_b)

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

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

```

## 视觉问答模型

当被问及关于图片的自然语言问题时，该模型可以选择正确的单词作答。

它通过将问题和图像编码成向量，然后连接两者，在上面训练一个逻辑回归，来从词汇表中挑选一个可能的单词作答。

```

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

# 首先，让我们用 Sequential 来定义一个视觉模型。
# 这个模型会把一张图像编码为向量。
vision_model = Sequential()
vision_model.add(Conv2D(64, (3, 3), activation='relu', padding='same',
input_shape=(224, 224, 3)))
vision_model.add(Conv2D(64, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
vision_model.add(Conv2D(128, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
vision_model.add(Conv2D(256, (3, 3), activation='relu'))
vision_model.add(Conv2D(256, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Flatten())

# 现在让我们用视觉模型来得到一个输出张量：
image_input = Input(shape=(224, 224, 3))
encoded_image = vision_model(image_input)

# 接下来，定义一个语言模型来将问题编码成一个向量。
# 每个问题最长 100 个词，词的索引从 1 到 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)

# 连接问题向量和图像向量：
merged = keras.layers.concatenate([encoded_question, encoded_image])

```

```
# 然后在上面训练一个 1000 词的逻辑回归模型:  
output = Dense(1000, activation='softmax')(merged)  
  
# 最终模型:  
vqa_model = Model(inputs=[image_input, question_input], outputs=output)  
  
# 下一步就是在真实数据上训练模型。
```

## 视频问答模型

现在我们已经训练了图像问答模型，我们可以很快地将它转换为视频问答模型。在适当的训练下，你可以给它展示一小段视频（例如 100 帧的人体动作），然后问它一个关于这段视频的问题（例如，「这个人在做什么运动？」 -> 「足球」）。

```
from keras.layers import TimeDistributed  
  
video_input = Input(shape=(100, 224, 224, 3))  
# 这是基于之前定义的视觉模型（权重被重用）构建的视频编码  
encoded_frame_sequence = TimeDistributed(vision_model)(video_input) # 输出为向量的序列  
encoded_video = LSTM(256)(encoded_frame_sequence) # 输出为一个向量  
  
# 这是问题编码器的模型级表示，重复使用与之前相同的权重：  
question_encoder = Model(inputs=question_input, outputs=encoded_question)  
  
# 让我们用它来编码这个问题：  
video_question_input = Input(shape=(100,), dtype='int32')  
encoded_video_question = question_encoder(video_question_input)  
  
# 这就是我们的视频问答模式：  
merged = keras.layers.concatenate([encoded_video, encoded_video_question])  
output = Dense(1000, activation='softmax')(merged)  
video_qa_model = Model(inputs=[video_input, video_question_input],  
outputs=output)
```



# Keras FAQ: 常见问题解答

- 如何引用 Keras?
- 如何在 GPU 上运行 Keras?
- 如何在多 GPU 上运行 Keras 模型?
- “sample”, “batch”, “epoch” 分别是什么?
- 如何保存 Keras 模型?
- 为什么训练集误差比测试集的误差高很多?
- 如何获取中间层的输出?
- 如何用 Keras 处理超过内存的数据集?
- 在验证集的误差不再下降时，如何中断训练?
- 验证集划分是如何计算的?
- 在训练过程中数据是否会混淆?
- 如何在每个 epoch 后记录训练集和验证集的误差和准确率?
- 如何「冻结」网络层?
- 如何使用状态 RNNs (stateful RNNs)?
- 如何从 Sequential 模型中移除一个层?
- 如何在 Keras 中使用预训练的模型?
- 如何在 Keras 中使用 HDF5 输入?
- Keras 配置文件保存在哪里?
- 如何在 Keras 开发过程中获取可复现的结果?
- 如何在 Keras 中安装 HDF5 或 h5py 来保存我的模型?

## 如何引用 Keras?

如果 Keras 有助于您的研究，请在你的出版物中引用它。以下是 BibTeX 条目引用的示例：

```
@misc{chollet2015keras,  
    title={Keras},  
    author={Chollet, Fran\c{c}ois and others},  
    year={2015},  
    publisher={GitHub},
```

```
    howpublished={\url{https://github.com/keras-team/keras}}},  
}
```

## 如何在 GPU 上运行 Keras?

如果你以 **TensorFlow** 或 **CNTK** 后端运行，只要检测到任何可用的 GPU，那么代码将自动在 GPU 上运行。

如果你以 **Theano** 后端运行，则可以使用以下方法之一：

**方法 1:** 使用 Theano flags。

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

“gpu”可能需要根据你的设备标识符（例如gpu0, gpu1等）进行更改。

**方法 2:** 创建 `.theanorc`: [指导教程](#)

**方法 3:** 在代码的开头手动设置 `theano.config.device`, `theano.config.floatX`:

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

## 如何在多 GPU 上运行 Keras 模型?

我们建议使用 **TensorFlow** 后端来执行这项任务。有两种方法可在多个 GPU 上运行单个模型：  
**数据并行\*\*和\*\*设备并行。**

在大多数情况下，你最需要的是数据并行。

### 数据并行

数据并行包括在每个设备上复制一次目标模型，并使用每个模型副本处理不同部分的输入数据。Keras 有一个内置的实用函数 `keras.utils.multi_gpu_model`，它可以生成任何模型的数据并行版本，在多达 8 个 GPU 上实现准线性加速。

有关更多信息，请参阅 [multi\\_gpu\\_model](#) 的文档。这里是一个快速的例子：

```
from keras.utils import multi_gpu_model  
  
# 将 `model` 复制到 8 个 GPU 上。
```

```
# 假定你的机器有 8 个可用的 GPU。
parallel_model = multi_gpu_model(model, gpus=8)
parallel_model.compile(loss='categorical_crossentropy',
                       optimizer='rmsprop')

# 这个 `fit` 调用将分布在 8 个 GPU 上。
# 由于 batch size 为 256，每个 GPU 将处理 32 个样本。
parallel_model.fit(x, y, epochs=20, batch_size=256)
```

## 设备并行

设备并行性包括在不同设备上运行同一模型的不同部分。对于具有并行体系结构的模型，例如有两个分支的模型，这种方式很合适。

这种并行可以通过使用 TensorFlow device scopes 来实现。这里是一个简单的例子：

```
# 模型中共享的 LSTM 用于并行编码两个不同的序列
input_a = keras.Input(shape=(140, 256))
input_b = keras.Input(shape=(140, 256))

shared_lstm = keras.layers.LSTM(64)

# 在一个 GPU 上处理第一个序列
with tf.device_scope('/gpu:0'):
    encoded_a = shared_lstm(tweet_a)
# 在另一个 GPU 上处理下一个序列
with tf.device_scope('/gpu:1'):
    encoded_b = shared_lstm(tweet_b)

# 在 CPU 上连接结果
with tf.device_scope('/cpu:0'):
    merged_vector = keras.layers.concatenate([encoded_a, encoded_b],
                                              axis=-1)
```

## “sample”, “batch”, “epoch” 分别是什么？

为了正确地使用 Keras，以下是必须了解和理解的一些常见定义：

- **Sample:** 样本，数据集中的一个元素，一条数据。
  - **例1:** 在卷积神经网络中，一张图像是一个样本。
  - **例2:** 在语音识别模型中，一段音频是一个样本。
- **Batch:** 批次，含有  $N$  个样本的集合。每一个 batch 的样本都是独立并行处理的。在训练时，一个 batch 的结果只会用来更新一次模型。
  - 一个 **batch** 的样本通常比单个输入更接近于总体输入数据的分布，batch 越大就越相似。然而，每个 batch 将花费更长的时间来处理，并且仍然只更新模型一次。在推理

(评估/预测) 时，建议条件允许的情况下选择一个尽可能大的 batch，(因为较大的 batch 通常评估/预测的速度会更快)。

- **Epoch**: 轮次，通常被定义为「在整个数据集上的一轮迭代」，用于训练的不同的阶段，这有利于记录和定期评估。
  - 当在 Keras 模型的 `fit` 方法中使用 `validation_data` 或 `validation_split` 时，评估将在每个 **epoch** 结束时运行。
  - 在 Keras 中，可以添加专门的用于在 **epoch** 结束时运行的 `callbacks` 回调。例如学习率变化和模型检查点（保存）。

## 如何保存 Keras 模型？

### 保存/加载整个模型（结构 + 权重 + 优化器状态）

不建议使用 `pickle` 或 `cPickle` 来保存 Keras 模型。

你可以使用 `model.save(filepath)` 将 Keras 模型保存到单个 HDF5 文件中，该文件将包含：

- 模型的结构，允许重新创建模型
- 模型的权重
- 训练配置项（损失函数，优化器）
- 优化器状态，允许准确地从你上次结束的地方继续训练。

你可以使用 `keras.models.load_model(filepath)` 重新实例化模型。`load_model` 还将负责使用保存的训练配置项来编译模型（除非模型从未编译过）。

示例：

```
from keras.models import load_model

model.save('my_model.h5') # 创建 HDF5 文件 'my_model.h5'
del model # 删除现有模型

# 返回一个编译好的模型
# 与之前那个相同
model = load_model('my_model.h5')
```

另请参阅[如何安装 HDF5 或 h5py 以在 Keras 中保存我的模型](#)，查看有关如何安装 h5py 的说明。

### 只保存/加载模型的结构

如果您只需要保存\*\*模型的结构\*\*，而非其权重或训练配置项，则可以执行以下操作：

```
# 保存为 JSON  
json_string = model.to_json()  
  
# 保存为 YAML  
yaml_string = model.to_yaml()
```

生成的 JSON/YAML 文件是人类可读的，如果需要还可以手动编辑。

你可以从这些数据建立一个新的模型：

```
# 从 JSON 重建模型:  
from keras.models import model_from_json  
model = model_from_json(json_string)  
  
# 从 YAML 重建模型:  
from keras.models import model_from_yaml  
model = model_from_yaml(yaml_string)
```

## 只保存/加载模型的权重

如果您只需要 **模型的权重**，可以使用下面的代码以 HDF5 格式进行保存。

请注意，我们首先需要安装 HDF5 和 Python 库 h5py，它们不包含在 Keras 中。

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

假设你有用于实例化模型的代码，则可以将保存的权重加载到具有相同结构的模型中：

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

如果你需要将权重加载到不同的结构（有一些共同层）的模型中，例如微调或迁移学习，则可以按层的名字来加载权重：

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

示例：

```
....  
假设原始模型如下所示：  
model = Sequential()  
model.add(Dense(2, input_dim=3, name='dense_1'))  
model.add(Dense(3, name='dense_2'))  
...  
model.save_weights(fname)  
....  
  
# 新模型  
model = Sequential()  
model.add(Dense(2, input_dim=3, name='dense_1')) # 将被加载
```

```
model.add(Dense(10, name='new_dense')) # 将不被加载  
  
# 从第一个模型加载权重；只会影响第一层，dense_1  
model.load_weights(fname, by_name=True)
```

### 处理已保存模型中的自定义层（或其他自定义对象）

如果要加载的模型包含自定义层或其他自定义类或函数，则可以通过 `custom_objects` 参数将它们传递给加载机制：

```
from keras.models import load_model  
# 假设你的模型包含一个 AttentionLayer 类的实例  
model = load_model('my_model.h5', custom_objects={'AttentionLayer':  
AttentionLayer})
```

或者，你可以使用自定义对象作用域：

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

自定义对象的处理与 `load_model`, `model_from_json`, `model_from_yaml` 的工作方式相同：

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

### 为什么训练误差比测试误差高很多？

Keras 模型有两种模式：训练和测试。正则化机制，如 Dropout 和 L1/L2 权重正则化，在测试时是关闭的。

此外，训练误差是每批训练数据的平均误差。由于你的模型是随着时间而变化的，一个 epoch 中的第一批数据的误差通常比最后一批的要高。另一方面，测试误差是模型在一个 epoch 训练完后计算的，因而误差较小。

### 如何获取中间层的输出？

一个简单的方法是创建一个新的 `Model` 来输出你所感兴趣的层：

```
from keras.models import Model

model = ... # 创建原始模型

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

或者，你也可以构建一个 Keras 函数，该函数将在给定输入的情况下返回某个层的输出，例如：

```
from keras import backend as K

# 以 Sequential 模型为例
get_3rd_layer_output = K.function([model.layers[0].input],
                                  [model.layers[3].output])
layer_output = get_3rd_layer_output([x])[0]
```

同样，你可以直接建立一个 Theano 或 TensorFlow 函数。

注意，如果你的模型在训练和测试阶段有不同的行为（例如，使用 Dropout，BatchNormalization 等），则需要将学习阶段标志传递给你的函数：

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

# 测试模式 = 0 时的输出
layer_output = get_3rd_layer_output([x, 0])[0]

# 测试模式 = 1 时的输出
layer_output = get_3rd_layer_output([x, 1])[0]
```

## 如何用 Keras 处理超过内存的数据集？

你可以使用 `model.train_on_batch(x, y)` 和 `model.test_on_batch(x, y)` 进行批量训练与测试。请参阅[模型文档](#)。

或者，你可以编写一个生成批处理训练数据的生成器，然后使用  
`model.fit_generator(data_generator, steps_per_epoch, epochs)` 方法。

你可以在[CIFAR10 example](#) 中找到实践代码。

## 在验证集的误差不再下降时，如何中断训练？

你可以使用 `EarlyStopping` 回调：

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

更多信息请查看 [callbacks 文档](#)。

## 验证集划分是如何计算的？

如果您将 `model.fit` 中的 `validation_split` 参数设置为 0.1，那么使用的验证数据将是最后 10% 的数据。如果设置为 0.25，就是最后 25% 的数据。注意，在提取分割验证集之前，数据不会被混洗，因此验证集仅仅是传递的输入中最后一个 x% 的样本。

所有 epoch 都使用相同的验证集（在同一个 `fit` 中调用）。

## 在训练过程中数据是否会混洗？

是的，如果 `model.fit` 中的 `shuffle` 参数设置为 `True`（默认值），则训练数据将在每个 epoch 混洗。

验证集永远不会混洗。

## 如何在每个 epoch 后记录训练集和验证集的误差和准确率？

`model.fit` 方法返回一个 `History` 回调，它具有包含连续误差的列表和其他度量的 `history` 属性。

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

## 如何「冻结」网络层？

「冻结」一个层意味着将其排除在训练之外，即其权重将永远不会更新。这在微调模型或使用固定的词向量进行文本输入中很有用。

您可以将 `trainable` 参数（布尔值）传递给一个层的构造器，以将该层设置为不可训练的：

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

另外，可以在实例化之后将网络层的 `trainable` 属性设置为 `True` 或 `False`。为了使之生效，在修改 `trainable` 属性之后，需要在模型上调用 `compile()`。这是一个例子：

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

frozen_model = Model(x, y)
# 在下面的模型中，训练期间不会更新层的权重
frozen_model.compile(optimizer='rmsprop', loss='mse')

layer.trainable = True
trainable_model = Model(x, y)
# 使用这个模型，训练期间 `layer` 的权重将被更新
# (这也会影响上面的模型，因为它使用了同一个网络层实例)
trainable_model.compile(optimizer='rmsprop', loss='mse')

frozen_model.fit(data, labels) # 这不会更新 `layer` 的权重
trainable_model.fit(data, labels) # 这会更新 `layer` 的权重
```

## 如何使用有状态 RNN (stateful RNNs)？

使 RNN 具有状态意味着每批样品的状态将被重新用作下一批样品的初始状态。

当使用有状态 RNN 时，假定：

- 所有的批次都有相同数量的样本
- 如果 `x1` 和 `x2` 是连续批次的样本，则 `x2[i]` 是 `x1[i]` 的后续序列，对于每个 `i`。

要在 RNN 中使用状态，你需要：

- 通过将 `batch_size` 参数传递给模型的第一层来显式指定你正在使用的批大小。例如，对于 10 个时间步长的 32 样本的 batch，每个时间步长具有 16 个特征，`batch_size = 32`。
- 在 RNN 层中设置 `stateful = True`。
- 在调用 `fit()` 时指定 `shuffle = False`。

重置累积状态：

- 使用 `model.reset_states()` 来重置模型中所有层的状态
- 使用 `layer.reset_states()` 来重置指定有状态 RNN 层的状态

示例：

```
x # 输入数据, 尺寸为 (32, 21, 16)
# 将步长为 10 的序列输送到模型中

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

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

# 训练网络, 根据给定的前 10 个时间步, 来预测第 11 个时间步:
model.train_on_batch(x[:, :10, :], np.reshape(x[:, 10, :], (32, 16)))

# 网络的状态已经改变。我们可以提供后续序列:
model.train_on_batch(x[:, 10:20, :], np.reshape(x[:, 20, :], (32, 16)))

# 重置 LSTM 层的状态:
model.reset_states()

# 另一种重置方法:
model.layers[0].reset_states()
```

请注意, `predict`, `fit`, `train_on_batch`, `predict_classes` 等方法\*全部\*都会更新模型中有状态层的状态。这使你不仅可以进行有状态的训练, 还可以进行有状态的预测。

## 如何从 Sequential 模型中移除一个层?

你可以通过调用 `.pop()` 来删除 Sequential 模型中最后添加的层：

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

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

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

## 如何在 Keras 中使用预训练的模型?

我们提供了以下图像分类模型的代码和预训练的权重：

- Xception
- VGG16
- VGG19
- ResNet50
- ResNet v2
- ResNeXt
- Inception v3
- Inception-ResNet v2
- MobileNet v1
- MobileNet v2
- DenseNet
- NASNet

它们可以使用 `keras.applications` 模块进行导入：

```
from keras.applications.xception import Xception
from keras.applications.vgg16 import VGG16
from keras.applications.vgg19 import VGG19
from keras.applications.resnet50 import ResNet50
from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_resnet_v2 import InceptionResNetV2
from keras.applications.mobilenet import MobileNet
from keras.applications.densenet import DenseNet121
from keras.applications.densenet import DenseNet169
from keras.applications.densenet import DenseNet201
from keras.applications.nasnet import NASNetLarge
from keras.applications.nasnet import NASNetMobile
from keras.applications.mobilenet_v2 import MobileNetV2

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

有关一些简单的用法示例，请参阅 [Applications 模块的文档](#)。

有关如何使用此类预训练的模型进行特征提取或微调的详细示例，请参阅[此博客文章](#)。

VGG16 模型也是以下几个 Keras 示例脚本的基础：

- [Style transfer](#)
- [Feature visualization](#)
- [Deep dream](#)

## 如何在 Keras 中使用 HDF5 输入？

你可以使用 `keras.utils` 中的 `HDF5Matrix` 类。有关详细信息，请参阅 [HDF5Matrix 文档](#)。

你也可以直接使用 HDF5 数据集：

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

请查看[如何在 Keras 中安装 HDF5 或 h5py 来保存模型](#)找到 h5py 安装指引。

## Keras 配置文件保存在哪里？

所有 Keras 数据存储的默认目录是：

```
$HOME/.keras/
```

注意，Windows 用户应该将 `$HOME` 替换为 `%USERPROFILE%`。如果 Keras 无法创建上述目录（例如，由于权限问题），则使用 `/tmp/.keras/` 作为备份。

Keras 配置文件是存储在 `$HOME/.keras/keras.json` 中的 JSON 文件。默认的配置文件如下所示：

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

它包含以下字段：

- 图像处理层和实用程序所使用的默认值图像数据格式（`channels_last` 或 `channels_first`）。
- 用于防止在某些操作中被零除的 `epsilon` 模糊因子。
- 默认浮点数据类型。
- 默认后端。详见 [backend 文档](#)。

同样，缓存的数据集文件（如使用 `get_file()` 下载的文件）默认存储在 `$HOME/.keras/datasets/` 中。

## 如何在 Keras 开发过程中获取可复现的结果？

在模型的开发过程中，能够在一次次的运行中获得可复现的结果，以确定性能的变化是来自模型还是数据集的变化，或者仅仅是一些新的随机样本点带来的结果，有时候是很有用处的。

首先，你需要在程序启动之前将 `PYTHONHASHSEED` 环境变量设置为 0（不在程序本身内）。对于 Python 3.2.3 以上版本，它对于某些基于散列的操作具有可重现的行为是必要的（例如，集合和字典的 item 顺序，请参阅 [Python 文档](#) 和 [issue #2280](#) 获取更多详细信息）。设置环境变量的一种方法是在这样启动 python 时：

```
$ cat test_hash.py
print(hash("keras"))
$ python3 test_hash.py          # 无法复现的 hash (Python 3.2.3+)
-8127205062320133199
$ python3 test_hash.py          # 无法复现的 hash (Python 3.2.3+)
3204480642156461591
$ PYTHONHASHSEED=0 python3 test_hash.py # 可复现的 hash
4883664951434749476
$ PYTHONHASHSEED=0 python3 test_hash.py # 可复现的 hash
4883664951434749476
```

此外，当使用 TensorFlow 后端并在 GPU 上运行时，某些操作具有非确定性输出，特别是 `tf.reduce_sum()`。这是因为 GPU 并行运行许多操作，因此并不总能保证执行顺序。由于浮点数的精度有限，即使添加几个数字，也可能会产生略有不同的结果，具体取决于添加它们的顺序。你可以尝试避免某些非确定性操作，但有些操作可能是由 TensorFlow 在计算梯度时自动创建的，因此在 CPU 上运行代码要简单得多。为此，你可以将 `CUDA_VISIBLE_DEVICES` 环境变量设置为空字符串，例如：

```
$ CUDA_VISIBLE_DEVICES="" PYTHONHASHSEED=0 python your_program.py
```

下面的代码片段提供了一个如何获得可复现结果的例子 - 针对 Python 3 环境的 TensorFlow 后端。

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

# 以下是 Numpy 在一个明确的初始状态生成固定随机数字所必需的。
np.random.seed(42)

# 以下是 Python 在一个明确的初始状态生成固定随机数字所必需的。
rn.seed(12345)

# 强制 TensorFlow 使用单线程。
# 多线程是结果不可复现的一个潜在因素。
# 更多详情，见：https://stackoverflow.com/questions/42022950/

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

from keras import backend as K

# `tf.set_random_seed()` 将会以 TensorFlow 为后端，
# 在一个明确的初始状态下生成固定随机数字。
# 更多详情，见：https://www.tensorflow.org/api\_docs/python/tf/set\_random\_seed

tf.set_random_seed(1234)

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

```
# 剩余代码 ...
```

## 如何在 Keras 中安装 HDF5 或 h5py 来保存我的模型?

为了将你的 Keras 模型保存为 HDF5 文件，例如通过 `keras.callbacks.ModelCheckpoint`，Keras 使用了 h5py Python 包。h5py 是 Keras 的依赖项，应默认被安装。在基于 Debian 的发行版本上，你需要再额外安装 `libhdf5`：

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

如果你不确定是否安装了 h5py，则可以打开 Python shell 并通过下面的命令加载模块

```
import h5py
```

如果模块导入没有错误，那么说明模块已经安装成功，否则你可以在 <http://docs.h5py.org/en/latest/build.html> 中找到详细的安装说明。



# 关于 Keras 模型

在 Keras 中有两类主要的模型：[Sequential 顺序模型](#) 和 [使用函数式 API 的 Model 类模型](#)。

这些模型有许多共同的方法和属性：

- `model.layers` 是包含模型网络层的展平列表。
- `model.inputs` 是模型输入张量的列表。
- `model.outputs` 是模型输出张量的列表。
- `model.summary()` 打印出模型概述信息。它是 [utils.print\\_summary](#) 的简捷调用。
- `model.get_config()` 返回包含模型配置信息的字典。通过以下代码，就可以根据这些配置信息重新实例化模型：

```
config = model.get_config()
model = Model.from_config(config)
# 或者, 对于 Sequential:
model = Sequential.from_config(config)
```

- `model.get_weights()` 返回模型中所有权重张量的列表，类型为 Numpy 数组。
- `model.set_weights(weights)` 从 Numpy 数组中为模型设置权重。列表中的数组必须与 `get_weights()` 返回的权重具有相同的尺寸。
- `model.to_json()` 以 JSON 字符串的形式返回模型的表示。请注意，该表示不包括权重，仅包含结构。你可以通过以下方式从 JSON 字符串重新实例化同一模型（使用重新初始化的权重）：

```
from keras.models import model_from_json

json_string = model.to_json()
model = model_from_json(json_string)
```

- `model.to_yaml()` 以 YAML 字符串的形式返回模型的表示。请注意，该表示不包括权重，只包含结构。你可以通过以下代码，从 YAML 字符串中重新实例化相同的模型（使用重新初始化的权重）：

```
from keras.models import model_from_yaml

yaml_string = model.to_yaml()
model = model_from_yaml(yaml_string)
```

- `model.save_weights(filepath)` 将模型权重存储为 HDF5 文件。

- `model.load_weights(filepath, by_name=False)`: 从 HDF5 文件（由 `save_weights` 创建）中加载权重。默认情况下，模型的结构应该是不变的。如果想将权重载入不同的模型（部分层相同），设置 `by_name=True` 来载入那些名字相同的层的权重。

注意：另请参阅[如何安装 HDF5 或 h5py 以保存 Keras 模型](#)，在常见问题中了解如何安装 h5py 的说明。

## Model 类继承

除了这两类模型之外，你还可以通过继承 `Model` 类并在 `call` 方法中实现你自己的前向传播，以创建你自己的完全定制化的模型，（`Model` 类继承 API 引入于 Keras 2.2.0）。

这里是一个用 `Model` 类继承写的简单的多层感知器的例子：

```
import keras

class SimpleMLP(keras.Model):

    def __init__(self, use_bn=False, use_dp=False, num_classes=10):
        super(SimpleMLP, self).__init__(name='mlp')
        self.use_bn = use_bn
        self.use_dp = use_dp
        self.num_classes = num_classes

        self.dense1 = keras.layers.Dense(32, activation='relu')
        self.dense2 = keras.layers.Dense(num_classes, activation='softmax')
        if self.use_dp:
            self.dp = keras.layers.Dropout(0.5)
        if self.use_bn:
            self.bn = keras.layers.BatchNormalization(axis=-1)

    def call(self, inputs):
        x = self.dense1(inputs)
        if self.use_dp:
            x = self.dp(x)
        if self.use_bn:
            x = self.bn(x)
        return self.dense2(x)

model = SimpleMLP()
model.compile(...)
model.fit(...)
```

网络层定义在 `__init__(self, ...)` 中，前向传播在 `call(self, inputs)` 中指定。在 `call` 中，你可以指定自定义的损失函数，通过调用 `self.add_loss(loss_tensor)`（就像你在自定义层中一样）。

在类继承模型中，模型的拓扑结构是由 Python 代码定义的（而不是网络层的静态图）。这意味着该模型的拓扑结构不能被检查或序列化。因此，以下方法和属性\*\*不适用于类继承模型\*\*：

- `model.inputs` 和 `model.outputs`。
- `model.to_yaml()` 和 `model.to_json()`。
- `model.get_config()` 和 `model.save()`。

**关键点：**为每个任务使用正确的 API。`Model` 类继承 API 可以为实现复杂模型提供更大的灵活性，但它需要付出代价（比如缺失的特性）：它更冗长，更复杂，并且有更多的用户错误机会。如果可能的话，尽可能使用函数式 API，这对用户更友好。



# Sequential 模型 API

在阅读这片文档前, 请先阅读 [Keras Sequential 模型指引](#)。

## Sequential 模型方法

### compile

```
compile(optimizer, loss=None, metrics=None, loss_weights=None,  
sample_weight_mode=None, weighted_metrics=None, target_tensors=None)
```

用于配置训练模型。

#### 参数

- **optimizer**: 字符串（优化器名）或者优化器对象。详见 [optimizers](#)。
- **loss**: 字符串（目标函数名）或目标函数或 `Loss` 实例。详见 [losses](#)。如果模型具有多个输出, 则可以通过传递损失函数的字典或列表, 在每个输出上使用不同的损失。模型将最小化的损失值将是所有单个损失的总和。
- **metrics**: 在训练和测试期间的模型评估标准。通常你会使用 `metrics = ['accuracy']`。要为多输出模型的不同输出指定不同的评估标准, 还可以传递一个字典, 如  
`metrics={'output_a': 'accuracy', 'output_b': ['accuracy', 'mse']}`。你也可以传递一个评估指标序列的序列 (`len = len(outputs)`) 例如 `metrics=[[['accuracy'], ['accuracy', 'mse']]]` 或 `metrics=['accuracy', ['accuracy', 'mse']]`。
- **loss\_weights**: 指定标量系数（Python浮点数）的可选列表或字典, 用于加权不同模型输出的损失贡献。模型将要最小化的损失值将是所有单个损失的加权和, 由 `loss_weights` 系数加权。如果是列表, 则期望与模型的输出具有 1:1 映射。如果是字典, 则期望将输出名称（字符串）映射到标量系数。
- **sample\_weight\_mode**: 如果你需要执行按时间步采样权重（2D 权重）, 请将其设置为 `temporal`。默认为 `None`, 为采样权重（1D）。如果模型有多个输出, 则可以通过传递 `mode` 的字典或列表, 以在每个输出上使用不同的 `sample_weight_mode`。
- **weighted\_metrics**: 在训练和测试期间, 由 `sample_weight` 或 `class_weight` 评估和加权的度量标准列表。
- **target\_tensors**: 默认情况下, Keras 将为模型的目标创建一个占位符, 在训练过程中将使用目标数据。相反, 如果你想使用自己的目标张量（反过来说, Keras 在训练期间不会载入这些目标张量的外部 Numpy 数据）, 您可以通过 `target_tensors` 参数指定它们。它应该是单个张量（对于单输出 Sequential 模型）。

- **\*\*kwargs**: 当使用 Theano/CNTK 后端时，这些参数被传入 `K.function`。当使用 TensorFlow 后端时，这些参数被传递到 `tf.Session.run`。

## 异常

- **ValueError**: 如果 `optimizer`, `loss`, `metrics` 或 `sample_weight_mode` 这些参数不合法。

## fit

```
fit(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, validation_freq=1, max_queue_size=10, workers=1,
use_multiprocessing=False)
```

以固定数量的轮次（数据集上的迭代）训练模型。

## 参数

- **x**: 输入数据。可以是：
  - 一个 Numpy 数组（或类数组），或者数组的序列（如果模型有多个输入）。
  - 一个将名称匹配到对应数组/张量的字典，如果模型具有命名输入。
  - 一个返回 `(inputs, targets)` 或 `(inputs, targets, sample weights)` 的生成器或 `keras.utils.Sequence`。
  - `None`（默认），如果从本地框架张量馈送（例如 TensorFlow 数据张量）。
- **y**: 目标数据。与输入数据 `x` 类似，它可以是 Numpy 数组（序列）、本地框架张量（序列）、Numpy 数组序列（如果模型有多个输出）或 `None`（默认）如果从本地框架张量馈送（例如 TensorFlow 数据张量）。如果模型输出层已命名，你也可以传递一个名称匹配 Numpy 数组的字典。如果 `x` 是一个生成器，或 `keras.utils.Sequence` 实例，则不应该指定 `y`（因为目标可以从 `x` 获得）。
- **batch\_size**: 整数或 `None`。每次梯度更新的样本数。如果未指定，默认为 32。如果你的数据是符号张量、生成器或 `Sequence` 实例形式，不要指定 `batch_size`，因为它们会生成批次。
- **epochs**: 整数。训练模型迭代轮次。一个轮次是在整个 `x` 或 `y` 上的一轮迭代。请注意，与 `initial_epoch` 一起，`epochs` 被理解为「最终轮次」。模型并不是训练了 `epochs` 轮，而是到第 `epochs` 轮停止训练。
- **verbose**: 整数，0, 1 或 2。日志显示模式。0 = 安静模式, 1 = 进度条, 2 = 每轮一行。
- **callbacks**: 一系列的 `keras.callbacks.Callback` 实例。一系列可以在训练和验证（如果有）时使用的回调函数。详见 [callbacks](#)。

- **validation\_split**: 0 和 1 之间的浮点数。用作验证集的训练数据的比例。模型将分出一部分不会被训练的验证数据，并将在每一轮结束时评估这些验证数据的误差和任何其他模型指标。验证数据是混洗之前 `x` 和 `y` 数据的最后一部分样本中。这个参数在 `x` 是生成器或 `Sequence` 实例时不支持。
- **validation\_data**: 用于在每个轮次结束后评估损失和任意指标的数据。模型不会在这个数据上训练。`validation_data` 会覆盖 `validation_split`。`validation_data` 可以是：
  - 元组 (`x_val`, `y_val`) 或 Numpy 数组或张量
  - 元组 (`x_val`, `y_val`, `val_sample_weights`) 或 Numpy 数组。
  - 数据集或数据集迭代器。

对于前两种情况，必须提供 `batch_size`。对于最后一种情况，必须提供 `validation_steps`。

- **shuffle**: 布尔值（是否在每轮迭代之前混洗数据）或者字符串（`batch`）。`batch` 是处理 HDF5 数据限制的特殊选项，它对一个 `batch` 内部的数据进行混洗。当 `steps_per_epoch` 非 `None` 时，这个参数无效。
- **class\_weight**: 可选的字典，用来映射类索引（整数）到权重（浮点）值，用于加权损失函数（仅在训练期间）。这可能有助于告诉模型「更多关注」来自代表性不足的类的样本。
- **sample\_weight**: 训练样本的可选 Numpy 权重数组，用于对损失函数进行加权（仅在训练期间）。你可以传递与输入样本长度相同的平坦（1D）Numpy 数组（权重和样本之间的 1:1 映射），或者在时序数据的情况下，可以传递尺寸为 (`samples`, `sequence_length`) 的 2D 数组，以对每个样本的每个时间步施加不同的权重。在这种情况下，你应该确保在 `compile()` 中指定 `sample_weight_mode="temporal"`。这个参数在 `x` 是生成器或 `Sequence` 实例时不支持，应该提供 `sample_weights` 作为 `x` 的第 3 元素。
- **initial\_epoch**: 整数。开始训练的轮次（有助于恢复之前的训练）。
- **steps\_per\_epoch**: 整数或 `None`。在声明一个轮次完成并开始下一个轮次之前的总步数（样本批次）。使用 TensorFlow 数据张量等输入张量进行训练时，默认值 `None` 等于数据集中样本的数量除以 `batch` 的大小，如果无法确定，则为 1。
- **validation\_steps**: 只有在提供了 `validation_data` 并且时一个生成器时才有用。表示在每个轮次结束时执行验证时，在停止之前要执行的步骤总数（样本批次）。
- **validation\_freq**: 只有在提供了验证数据时才有用。整数或列表/元组/集合。如果是整数，指定在新的验证执行之前要执行多少次训练，例如，`validation_freq=2` 在每 2 轮训练后执行验证。如果是列表、元组或集合，指定执行验证的轮次，例如，`validation_freq=[1, 2, 10]` 表示在第 1、2、10 轮训练后执行验证。
- **max\_queue\_size**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。生成器队列的最大尺寸。若未指定，`max_queue_size` 将默认为 10。
- **workers**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。当使用基于进程的多线程时的最大进程数。若未指定，`workers` 将默认为 1。若为 0，将在主线程执行生成器。

- **use\_multiprocessing**: 布尔值。仅用于生成器或 `keras.utils.Sequence` 输入。如果是 `True`，使用基于进程的多线程。若未指定，`use_multiprocessing` 将默认为 `False`。注意由于这个实现依赖于 `multiprocessing`，你不应该像生成器传递不可选的参数，因为它们不能轻松地传递给子进程。
- **\*\*kwargs**: 用于向后兼容。

## 返回

一个 `History` 对象。其 `History.history` 属性是连续 epoch 训练损失和评估值，以及验证集损失和评估值的记录（如果适用）。

## 异常

- **RuntimeError**: 如果模型从未编译。
- **ValueError**: 在提供的输入数据与模型期望的不匹配的情况下。

## evaluate

```
evaluate(x=None, y=None, batch_size=None, verbose=1, sample_weight=None,
steps=None, callbacks=None, max_queue_size=10, workers=1,
use_multiprocessing=False)
```

在测试模式，返回误差值和评估标准值。

计算逐批次进行。

## 参数

- **x**: 输入数据。可以是：
  - 一个 Numpy 数组（或类数组），或者数组的序列（如果模型有多个输入）。
  - 一个将名称匹配到对应数组/张量的字典，如果模型具有命名输入。
  - 一个返回 `(inputs, targets)` 或 `(inputs, targets, sample weights)` 的生成器或 `keras.utils.Sequence`。
  - `None`（默认），如果从本地框架张量馈送（例如 TensorFlow 数据张量）。
- **y**: 目标数据。与输入数据 `x` 类似，它可以是 Numpy 数组（序列）、本地框架张量（序列）、Numpy 数组序列（如果模型有多个输出）或 `None`（默认）如果从本地框架张量馈送（例如 TensorFlow 数据张量）。如果模型输出层已命名，你也可以传递一个名称匹配 Numpy 数组的字典。如果 `x` 是一个生成器，或 `keras.utils.Sequence` 实例，则不应该指定 `y`（因为目标可以从 `x` 获得）。

- **batch\_size**: 整数或 `None`。每次梯度更新的样本数。如果未指定，默認為 32。如果你的数据是符号张量、生成器或 `Sequence` 实例形式，不要指定 `batch_size`，因为它们会生成批次。
- **verbose**: 0, 1。日志显示模式。0 = 安静模式, 1 = 进度条。
- **sample\_weight**: 训练样本的可选 Numpy 权重数组，用于对损失函数进行加权。你可以传递与输入样本长度相同的平坦（1D）Numpy 数组（权重和样本之间的 1:1 映射），或者在时序数据的情况下，可以传递尺寸为 `(samples, sequence_length)` 的 2D 数组，以对每个样本的每个时间步施加不同的权重。在这种情况下，你应该确保在 `compile()` 中指定 `sample_weight_mode="temporal"`。
- **steps**: 整数或 `None`。声明评估结束之前的总步数（批次样本）。默認值 `None` 时被忽略。
- **callbacks**: 一系列的 `keras.callbacks.Callback` 实例。一系列可以在评估时使用的回调函数。详见 [callbacks](#)。
- **max\_queue\_size**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。生成器队列的最大尺寸。若未指定，`max_queue_size` 将默認為 10。
- **workers**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。当使用基于进程的多线程时的最大进程数。若未指定，`workers` 将默認為 1。若为 0，将在主线程执行生成器。
- **use\_multiprocessing**: 布尔值。仅用于生成器或 `keras.utils.Sequence` 输入。如果是 `True`，使用基于进程的多线程。若未指定，`use_multiprocessing` 将默認為 `False`。注意由于这个实现依赖于 `multiprocessing`，你不应该像生成器传递不可选的参数，因为它们不能轻松地传递给子进程。

## 异常

- **ValueError**: 若参数非法。

## 返回

标量测试误差（如果模型只有单个输出且没有评估指标）或标量列表（如果模型具有多个输出和/或指标）。属性 `model.metrics_names` 将提供标量输出的显示标签。

## predict

```
predict(x, batch_size=None, verbose=0, steps=None, callbacks=None,  
max_queue_size=10, workers=1, use_multiprocessing=False)
```

为输入样本生成输出预测。

计算逐批次进行。

## 参数

- **x**: 输入数据。可以是：
  - 一个 Numpy 数组（或类数组），或者数组的序列（如果模型有多个输入）。
  - 一个将名称匹配到对应数组/张量的字典，如果模型具有命名输入。
  - 一个返回 `(inputs, targets)` 或 `(inputs, targets, sample weights)` 的生成器或 `keras.utils.Sequence`。
  - `None`（默认），如果从本地框架张量馈送（例如 TensorFlow 数据张量）。
- **batch\_size**: 整数或 `None`。每次梯度更新的样本数。如果未指定，默認為 32。如果你的数据是符号张量、生成器或 `Sequence` 实例形式，不要指定 `batch_size`，因为它们会生成批次。
- **verbose**: 日志显示模式，0 或 1。
- **steps**: 声明预测结束之前的总步数（批次样本）。默認值 `None` 时被忽略。
- **callbacks**: 一系列的 `keras.callbacks.Callback` 实例。一系列可以在预测时使用的回调函数。详见 [callbacks](#)。
- **max\_queue\_size**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。生成器队列的最大尺寸。若未指定，`max_queue_size` 将默認為 10。
- **workers**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。当使用基于进程的多线程时的最大进程数。若未指定，`workers` 将默認為 1。若为 0，将在主线程执行生成器。
- **use\_multiprocessing**: 布尔值。仅用于生成器或 `keras.utils.Sequence` 输入。如果是 `True`，使用基于进程的多线程。若未指定，`use_multiprocessing` 将默認為 `False`。注意由于这个实现依赖于 `multiprocessing`，你不应该像生成器传递不可选的参数，因为它们不能轻松地传递给子进程。

## 返回

预测的 Numpy 数组。

## 异常

- **ValueError**: 如果提供的输入数据与模型的期望数据不匹配，或者有状态模型收到的数量不是批量大小的倍数。

## train\_on\_batch

```
train_on_batch(x, y, sample_weight=None, class_weight=None,  
reset_metrics=True)
```

一批样品的单次梯度更新。

## Arguments

- **x**: 训练数据的 Numpy 数组（如果模型只有一个输入），或者是 Numpy 数组的列表（如果模型有多个输入）。如果模型中的输入层被命名，你也可以传递一个字典，将输入层名称映射到 Numpy 数组。
- **y**: 目标（标签）数据的 Numpy 数组，或 Numpy 数组的列表（如果模型具有多个输出）。如果模型中的输出层被命名，你也可以传递一个字典，将输出层名称映射到 Numpy 数组。
- **sample\_weight**: 可选数组，与 x 长度相同，包含应用到模型损失函数的每个样本的权重。如果是时域数据，你可以传递一个尺寸为 (samples, sequence\_length) 的 2D 数组，为每一个样本的每一个时间步应用不同的权重。在这种情况下，你应该在 `compile()` 中指定 `sample_weight_mode="temporal"`。
- **class\_weight**: 可选的字典，用来映射类索引（整数）到权重（浮点）值，以在训练时对模型的损失函数加权。这可能有助于告诉模型「更多关注」来自代表性不足的类的样本。
- **reset\_metrics**: 如果为 `True`，返回的指标仅适用于该批次。如果为 `False`，则指标将在批次之间有状态地累积。

## 返回

标量训练误差（如果模型只有单个输出且没有评估指标）或标量列表（如果模型具有多个输出和/或指标）。属性 `model.metrics_names` 将提供标量输出的显示标签。

## test\_on\_batch

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

在一批样本上评估模型。

## 参数

- **x**: 测试数据的 Numpy 数组（如果模型只有一个输入），或者是 Numpy 数组的列表（如果模型有多个输入）。如果模型中的输入层被命名，你也可以传递一个字典，将输入层名称映射到 Numpy 数组。
- **y**: 目标（标签）数据的 Numpy 数组，或 Numpy 数组的列表（如果模型具有多个输出）。如果模型中的输出层被命名，你也可以传递一个字典，将输出层名称映射到 Numpy 数组。
- **sample\_weight**: 可选数组，与 x 长度相同，包含应用到模型损失函数的每个样本的权重。如果是时域数据，你可以传递一个尺寸为 (samples, sequence\_length) 的 2D 数组，为每一个样本的每一个时间步应用不同的权重。
- **reset\_metrics**: 如果为 `True`，返回的指标仅适用于该批次。如果为 `False`，则指标将在批次之间有状态地累积。

## 返回

标量测试误差（如果模型只有单个输出且没有评估指标）或标量列表（如果模型具有多个输出和/或指标）。属性 `model.metrics_names` 将提供标量输出的显示标签。

## `predict_on_batch`

```
predict_on_batch(x)
```

返回一批样本的模型预测值。

### 参数

- `x`: 输入数据，Numpy 数组或列表（如果模型有多输入）。

## 返回

预测值的 Numpy 数组。

## `fit_generator`

```
fit_generator(generator, steps_per_epoch=None, epochs=1, verbose=1,
              callbacks=None, validation_data=None, validation_steps=None,
              validation_freq=1, class_weight=None, max_queue_size=10, workers=1,
              use_multiprocessing=False, shuffle=True, initial_epoch=0)
```

使用 Python 生成器或 `Sequence` 实例逐批生成的数据，按批次训练模型。

生成器与模型并行运行，以提高效率。例如，这可以让你在 CPU 上对图像进行实时数据增强，以在 GPU 上训练模型。

`keras.utils.Sequence` 的使用可以保证数据的顺序，以及当 `use_multiprocessing=True` 时，保证每个输入在每个 epoch 只使用一次。

### 参数

- **generator**: 一个生成器，或者一个 `Sequence` (`keras.utils.Sequence`) 对象的实例，以在使用多进程时避免数据的重复。生成器的输出应该为以下之一：

- `(inputs, targets)` 元组
- `(inputs, targets, sample_weights)` 元组。

这个元组（生成器的单个输出）组成了单个的 batch。因此，这个元组中的所有数组长度必须相同（与这一个 batch 的大小相等）。不同的 batch 可能大小不同。例如，一个 epoch

的最后一个 batch 往往比其他 batch 要小，如果数据集的尺寸不能被 batch size 整除。生成器将无限地在数据集上循环。当运行到第 `steps_per_epoch` 时，记一个 epoch 结束。

- **`steps_per_epoch`**: 整数。在声明一个 epoch 完成并开始下一个 epoch 之前从 `generator` 产生的总步数（批次样本）。它通常应该等于 `ceil(num_samples / batch_size)`。对于 `Sequence`，它是可选的：如果未指定，将使用 `len(generator)` 作为步数。
- **`epochs`**: 整数。训练模型的迭代总轮数。一个 epoch 是对所提供的整个数据的一轮迭代，如 `steps_per_epoch` 所定义。注意，与 `initial_epoch` 一起使用，epoch 应被理解为「最后一轮」。模型没有经历由 `epochs` 给出的多次迭代的训练，而仅仅是直到达到索引 `epoch` 的轮次。
- **`verbose`**: 整数，0, 1 或 2。日志显示模式。0 = 安静模式, 1 = 进度条, 2 = 每轮一行。
- **`callbacks`**: 一系列的 `keras.callbacks.Callback` 实例。一系列可以在训练时使用的回调函数。详见 [callbacks](#)。
- **`validation_data`**: 它可以是以下之一：
  - 验证数据的生成器或 `Sequence` 实例
  - `(inputs, targets)` 元组
  - `(inputs, targets, sample_weights)` 元组。在每个 epoch 结束时评估损失和任何模型指标。该模型不会对此数据进行训练。
- **`validation_steps`**: 仅当 `validation_data` 是一个生成器时才可用。表示在每一轮迭代末尾停止前从 `validation_data` 生成地总步数（样本批次）。它应该等于由 batch size 分割的验证数据集的样本数。对于 `Sequence` 它是可选的：若未指定，将会使用 `len(validation_data)` 作为步数。
- **`validation_freq`**: 只有在提供了验证数据时才有用。整数或 `collections.Container` 实例（例如列表、元组等）。如果是整数，指定在新的验证执行之前要执行多少次训练，例如，`validation_freq=2` 在每 2 轮训练后执行验证。如果是 Container，指定执行验证的轮次，例如，`validation_freq=[1, 2, 10]` 表示在第 1、2、10 轮训练后执行验证。
- **`class_weight`**: 可选的将类索引（整数）映射到权重（浮点）值的字典，用于加权损失函数（仅在训练期间）。这可以用来告诉模型「更多地关注」来自代表性不足的类的样本。
- **`max_queue_size`**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。生成器队列的最大尺寸。若未指定，`max_queue_size` 将默认为 10。
- **`workers`**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。当使用基于进程的多线程时的最大进程数。若未指定，`workers` 将默认为 1。若为 0，将在主线程执行生成器。
- **`use_multiprocessing`**: 布尔值。仅用于生成器或 `keras.utils.Sequence` 输入。如果是 `True`，使用基于进程的多线程。若未指定，`use_multiprocessing` 将默认为 `False`。注意由于这个实现依赖于 multiprocessing，你不应该像生成器传递不可选的参数，因为它们不能轻松地传递给子进程。

- **shuffle**: 布尔值。是否在每轮迭代之前打乱 batch 的顺序。只能与 `Sequence` (`keras.utils.Sequence`) 实例同用。当 `steps_per_epoch` 为 `None` 是无效。
- **initial\_epoch**: 整数。开始训练的轮次（有助于恢复之前的训练）。

## 返回

一个 `History` 对象。其 `History.history` 属性是连续 epoch 训练损失和评估值，以及验证集损失和评估值的记录（如果适用）。

## 异常

- **ValueError**: 如果生成器生成的数据格式不正确。

## 示例

```
def generate_arrays_from_file(path):
    while True:
        with open(path) as f:
            for line in f:
                # 从文件中的每一行生成输入数据和标签的 numpy 数组
                x1, x2, y = process_line(line)
                yield ({'input_1': x1, 'input_2': x2}, {'output': y})

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

## evaluate\_generator

```
evaluate_generator(generator, steps=None, callbacks=None, max_queue_size=10,
workers=1, use_multiprocessing=False, verbose=0)
```

在数据生成器上评估模型。

这个生成器应该返回与 `test_on_batch` 所接收的同样的数据。

## 参数

- **generator**: 一个生成 `(inputs, targets)` 或 `(inputs, targets, sample_weights)` 的生成器，或一个 `Sequence` (`keras.utils.Sequence`) 对象的实例，以避免在使用多进程时数据的重复。
- **steps**: 在声明一个 epoch 完成并开始下一个 epoch 之前从 `generator` 产生的总步数（批次样本）。它通常应该等于你的数据集的样本数量除以批量大小。对于 `Sequence`，它是可选的：如果未指定，将使用 `len(generator)` 作为步数。
- **callbacks**: 一系列的 `keras.callbacks.Callback` 实例。一系列可以在评估时使用的回调函数。详见 [callbacks](#)。

- **max\_queue\_size**: 生成器队列的最大尺寸。
- **workers**: 整数。使用的最大进程数量，如果使用基于进程的多线程。如未指定，`workers` 将默认为 1。如果为 0，将在主线程上执行生成器。
- **use\_multiprocessing**: 布尔值。如果 True，则使用基于进程的多线程。请注意，由于此实现依赖于多进程，所以不应将不可传递的参数传递给生成器，因为它们不能被轻易地传递给子进程。
- **verbose**: 日志显示模式，0 或 1。

## 返回

标量测试误差（如果模型只有单个输出且没有评估指标）或标量列表（如果模型具有多个输出和/或指标）。属性 `model.metrics_names` 将提供标量输出的显示标签。

## 异常

- **ValueError**: 如果生成器生成的数据格式不正确。

## `predict_generator`

```
predict_generator(generator, steps=None, callbacks=None, max_queue_size=10,
                  workers=1, use_multiprocessing=False, verbose=0)
```

为来自数据生成器的输入样本生成预测。

这个生成器应该返回与 `predict_on_batch` 所接收的同样的数据。

## 参数

- **generator**: 生成器，返回批量输入样本，或一个 `Sequence` (`keras.utils.Sequence`) 对象的实例，以避免在使用多进程时数据的重复。
- **steps**: 在声明一个 epoch 完成并开始下一个 epoch 之前从 `generator` 产生的总步数（批次样本）。它通常应该等于你的数据集的样本数量除以批量大小。对于 `Sequence`，它是可选的：如果未指定，将使用 `len(generator)` 作为步数。
- **callbacks**: 一系列的 `keras.callbacks.Callback` 实例。一系列可以在预测时使用的回调函数。详见 [callbacks](#)。
- **max\_queue\_size**: 生成器队列的最大尺寸。
- **max\_queue\_size**: 生成器队列的最大尺寸。
- **workers**: 整数。使用的最大进程数量，如果使用基于进程的多线程。如未指定，`workers` 将默认为 1。如果为 0，将在主线程上执行生成器。

- **use\_multiprocessing**: 如果 True，则使用基于进程的多线程。请注意，由于此实现依赖于多进程，所以不应将不可传递的参数传递给生成器，因为它们不能被轻易地传递给子进程。
- **verbose**: 日志显示模式，0 或 1。

## 返回

预测值的 Numpy 数组。

## 异常

- **ValueError**: 如果生成器生成的数据格式不正确。

## get\_layer

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

根据名称（唯一）或索引值查找网络层。

如果同时提供了 name 和 index，则 index 将优先。

根据网络层的名称（唯一）或其索引返回该层。索引是基于水平图遍历的顺序（自下而上）。

## 参数

- **name**: 字符串，层的名字。
- **index**: 整数，层的索引。

## 返回

一个层实例。

## 异常

- **ValueError**: 如果层的名称或索引不正确。



# Model 类 (函数式 API)

在函数式 API 中，给定一些输入张量和输出张量，可以通过以下方式实例化一个 Model：

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

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

这个模型将包含从 a 到 b 的计算的所有网络层。

在多输入或多输出模型的情况下，你也可以使用列表：

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

有关 Model 的详细介绍，请阅读 [Keras 函数式 API 指引](#)。

## Model 类模型方法

### compile

```
compile(optimizer, loss=None, metrics=None, loss_weights=None,
sample_weight_mode=None, weighted_metrics=None, target_tensors=None)
```

用于配置训练模型。

#### 参数

- **optimizer**: 字符串（优化器名）或者优化器实例。详见 [optimizers](#)。
- **loss**: 字符串（目标函数名）或目标函数或 Loss 实例。详见 [losses](#)。如果模型具有多个输出，则可以通过传递损失函数的字典或列表，在每个输出上使用不同的损失。模型将最小化的损失值将是所有单个损失的总和。
- **metrics**: 在训练和测试期间的模型评估标准。通常你会使用 `metrics = ['accuracy']`。要为多输出模型的不同输出指定不同的评估标准，还可以传递一个字典，如  
`metrics={'output_a': 'accuracy', 'output_b': ['accuracy', 'mse']}`。你也可以传递一个评估指标序列的序列 (`len = len(outputs)`) 例如 `metrics=[[['accuracy'], ['accuracy', 'mse']]]` 或 `metrics=['accuracy', ['accuracy', 'mse']]`。
- **loss\_weights**: 可选的指定标量系数（Python 浮点数）的列表或字典，用以衡量损失函数对不同的模型输出的贡献。模型将最小化的误差值是由 `loss_weights` 系数加权的\*加权总和\*

误差。如果是列表，那么它应该是与模型输出相对应的 1:1 映射。如果是字典，那么应该把输出的名称（字符串）映到标量系数。

- **sample\_weight\_mode**: 如果你需要执行按时间步采样权重（2D 权重），请将其设置为 `temporal`。默认为 `None`，为采样权重（1D）。如果模型有多个输出，则可以通过传递 `mode` 的字典或列表，以在每个输出上使用不同的 `sample_weight_mode`。
- **weighted\_metrics**: 在训练和测试期间，由 `sample_weight` 或 `class_weight` 评估和加权的度量标准列表。
- **target\_tensors**: 默认情况下，Keras 将为模型的目标创建一个占位符，在训练过程中将使用目标数据。相反，如果你想使用自己的目标张量（反过来说，Keras 在训练期间不会载入这些目标张量的外部 Numpy 数据），您可以通过 `target_tensors` 参数指定它们。它可以是单个张量（单输出模型），张量列表，或一个映射输出名称到目标张量的字典。
- **\*\*kwargs**: 当使用 Theano/CNTK 后端时，这些参数被传入 `K.function`。当使用 TensorFlow 后端时，这些参数被传递到 `tf.Session.run`。

## 异常

- **ValueError**: 如果 `optimizer`, `loss`, `metrics` 或 `sample_weight_mode` 这些参数不合法。

## fit

```
fit(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, validation_freq=1, max_queue_size=10, workers=1,
use_multiprocessing=False)
```

以固定数量的轮次（数据集上的迭代）训练模型。

## 参数

- **x**: 输入数据。可以是：
  - 一个 Numpy 数组（或类数组），或者数组的序列（如果模型有多个输入）。
  - 一个将名称匹配到对应数组/张量的字典，如果模型具有命名输入。
  - 一个返回 `(inputs, targets)` 或 `(inputs, targets, sample weights)` 的生成器或 `keras.utils.Sequence`。
  - `None`（默认），如果从本地框架张量馈送（例如 TensorFlow 数据张量）。
- **y**: 目标数据。与输入数据 `x` 类似，它可以是 Numpy 数组（序列）、本地框架张量（序列）、Numpy 数组序列（如果模型有多个输出）或 `None`（默认）如果从本地框架张量馈送（例如 TensorFlow 数据张量）。如果模型输出层已命名，你也可以传递一个名称匹配

Numpy 数组的字典。如果 `x` 是一个生成器，或 `keras.utils.Sequence` 实例，则不应该指定 `y`（因为目标可以从 `x` 获得）。

- **batch\_size**: 整数或 `None`。每次梯度更新的样本数。如果未指定，默认为 32。如果你的数据是符号张量、生成器或 `Sequence` 实例形式，不要指定 `batch_size`，因为它们会生成批次。
- **epochs**: 整数。训练模型迭代轮次。一个轮次是在整个 `x` 和 `y` 上的一轮迭代。请注意，与 `initial_epoch` 一起，`epochs` 被理解为「最终轮次」。模型并不是训练了 `epochs` 轮，而是到第 `epochs` 轮停止训练。
- **verbose**: 整数，0, 1 或 2。日志显示模式。0 = 安静模式, 1 = 进度条, 2 = 每轮一行。
- **callbacks**: 一系列的 `keras.callbacks.Callback` 实例。一系列可以在训练和验证（如果有）时使用的回调函数。详见 [callbacks](#)。
- **validation\_split**: 0 和 1 之间的浮点数。用作验证集的训练数据的比例。模型将分出一部分不会被训练的验证数据，并将在每一轮结束时评估这些验证数据的误差和任何其他模型指标。验证数据是混洗之前 `x` 和 `y` 数据的最后一部分样本中。这个参数在 `x` 是生成器或 `Sequence` 实例时不支持。
- **validation\_data**: 用于在每个轮次结束后评估损失和任意指标的数据。模型不会在这个数据上训练。`validation_data` 会覆盖 `validation_split`。`validation_data` 可以是：
  - 元组 `(x_val, y_val)` 或 Numpy 数组或张量
  - 元组 `(x_val, y_val, val_sample_weights)` 或 Numpy 数组。
  - 数据集或数据集迭代器。
- 对于前两种情况，必须提供 `batch_size`。对于最后一种情况，必须提供 `validation_steps`。
- **shuffle**: 布尔值（是否在每轮迭代之前混洗数据）或者字符串 (`batch`)。`batch` 是处理 HDF5 数据限制的特殊选项，它对一个 `batch` 内部的数据进行混洗。当 `steps_per_epoch` 非 `None` 时，这个参数无效。
- **class\_weight**: 可选的字典，用来映射类索引（整数）到权重（浮点）值，用于加权损失函数（仅在训练期间）。这可能有助于告诉模型「更多关注」来自代表性不足的类的样本。
- **sample\_weight**: 训练样本的可选 Numpy 权重数组，用于对损失函数进行加权（仅在训练期间）。你可以传递与输入样本长度相同的平坦（1D）Numpy 数组（权重和样本之间的 1:1 映射），或者在时序数据的情况下，可以传递尺寸为 `(samples, sequence_length)` 的 2D 数组，以对每个样本的每个时间步施加不同的权重。在这种情况下，你应该确保在 `compile()` 中指定 `sample_weight_mode="temporal"`。这个参数在 `x` 是生成器或 `Sequence` 实例时不支持，应该提供 `sample_weights` 作为 `x` 的第 3 元素。
- **initial\_epoch**: 整数。开始训练的轮次（有助于恢复之前的训练）。

- **steps\_per\_epoch**: 整数或 `None`。在声明一个轮次完成并开始下一个轮次之前的总步数（样本批次）。使用 TensorFlow 数据张量等输入张量进行训练时，默认值 `None` 等于数据集中样本的数量除以 batch 的大小，如果无法确定，则为 1。
- **validation\_steps**: 只有在提供了 `validation_data` 并且时一个生成器时才有用。表示在每个轮次结束时执行验证时，在停止之前要执行的步骤总数（样本批次）。
- **validation\_freq**: 只有在提供了验证数据时才有用。整数或列表/元组/集合。如果是整数，指定在新的验证执行之前要执行多少次训练，例如，`validation_freq=2` 在每 2 轮训练后执行验证。如果是列表、元组或集合，指定执行验证的轮次，例如，`validation_freq=[1, 2, 10]` 表示在第 1、2、10 轮训练后执行验证。
- **max\_queue\_size**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。生成器队列的最大尺寸。若未指定，`max_queue_size` 将默认为 10。
- **workers**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。当使用基于进程的多线程时的最大进程数。若未指定，`workers` 将默认为 1。若为 0，将在主线程执行生成器。
- **use\_multiprocessing**: 布尔值。仅用于生成器或 `keras.utils.Sequence` 输入。如果是 `True`，使用基于进程的多线程。若未指定，`use_multiprocessing` 将默认为 `False`。注意由于这个实现依赖于 `multiprocessing`，你不应该像生成器传递不可选的参数，因为它们不能轻松地传递给子进程。
- **\*\*kwargs**: 用于向后兼容。

## 返回

一个 `History` 对象。其 `History.history` 属性是连续 epoch 训练损失和评估值，以及验证集损失和评估值的记录（如果适用）。

## 异常

- **RuntimeError**: 如果模型从未编译。
- **ValueError**: 在提供的输入数据与模型期望的不匹配的情况下。

## evaluate

```
evaluate(x=None, y=None, batch_size=None, verbose=1, sample_weight=None,
         steps=None, callbacks=None, max_queue_size=10, workers=1,
         use_multiprocessing=False)
```

在测试模式下返回模型的误差值和评估标准值。

计算是分批进行的。

## 参数

- **x**: 输入数据。可以是：
  - 一个 Numpy 数组（或类数组），或者数组的序列（如果模型有多个输入）。
  - 一个将名称匹配到对应数组/张量的字典，如果模型具有命名输入。
  - 一个返回 `(inputs, targets)` 或 `(inputs, targets, sample weights)` 的生成器或 `keras.utils.Sequence`。
  - `None`（默认），如果从本地框架张量馈送（例如 TensorFlow 数据张量）。
- **y**: 目标数据。与输入数据 **x** 类似，它可以是 Numpy 数组（序列）、本地框架张量（序列）、Numpy 数组序列（如果模型有多个输出）或 `None`（默认）如果从本地框架张量馈送（例如 TensorFlow 数据张量）。如果模型输出层已命名，你也可以传递一个名称匹配 Numpy 数组的字典。如果 **x** 是一个生成器，或 `keras.utils.Sequence` 实例，则不应该指定 **y**（因为目标可以从 **x** 获得）。
- **batch\_size**: 整数或 `None`。每次梯度更新的样本数。如果未指定，默認為 32。如果你的数据是符号张量、生成器或 `keras.utils.Sequence` 实例形式，不要指定 `batch_size`，因为它们会生成批次。
- **verbose**: 0, 1。日志显示模式。0 = 安静模式, 1 = 进度条。
- **sample\_weight**: 训练样本的可选 Numpy 权重数组，用于对损失函数进行加权。你可以传递与输入样本长度相同的平坦（1D）Numpy 数组（权重和样本之间的 1:1 映射），或者在时序数据的情况下，可以传递尺寸为 `(samples, sequence_length)` 的 2D 数组，以对每个样本的每个时间步施加不同的权重。在这种情况下，你应该确保在 `compile()` 中指定 `sample_weight_mode="temporal"`。
- **steps**: 整数或 `None`。声明评估结束之前的总步数（批次样本）。默认值 `None` 时被忽略。
- **callbacks**: 一系列的 `keras.callbacks.Callback` 实例。一系列可以在评估时使用的回调函数。详见 [callbacks](#)。
- **max\_queue\_size**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。生成器队列的最大尺寸。若未指定，`max_queue_size` 将默認為 10。
- **workers**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。当使用基于进程的多线程时的最大进程数。若未指定，`workers` 将默認為 1。若为 0，将在主线程执行生成器。
- **use\_multiprocessing**: 布尔值。仅用于生成器或 `keras.utils.Sequence` 输入。如果是 `True`，使用基于进程的多线程。若未指定，`use_multiprocessing` 将默認為 `False`。注意由于这个实现依赖于 `multiprocessing`，你不应该像生成器传递不可选的参数，因为它们不能轻松地传递给子进程。

## 异常

- **ValueError**: 如果参数错误。

## 返回

标量测试误差（如果模型只有一个输出且没有评估标准）或标量列表（如果模型具有多个输出和/或评估指标）。属性 `model.metrics_names` 将提供标量输出的显示标签。

## predict

```
predict(x, batch_size=None, verbose=0, steps=None, callbacks=None,
max_queue_size=10, workers=1, use_multiprocessing=False)
```

为输入样本生成输出预测。

计算是分批进行的

### 参数

- **x**: 输入数据。可以是：
  - 一个 Numpy 数组（或类数组），或者数组的序列（如果模型有多个输入）。
  - 一个将名称匹配到对应数组/张量的字典，如果模型具有命名输入。
  - 一个返回 `(inputs, targets)` 或 `(inputs, targets, sample weights)` 的生成器或 `keras.utils.Sequence`。
  - `None`（默认），如果从本地框架张量馈送（例如 TensorFlow 数据张量）。
- **batch\_size**: 整数或 `None`。每次梯度更新的样本数。如果未指定，默认为 32。如果你的数据是符号张量、生成器或 `keras.utils.Sequence` 实例形式，不要指定 `batch_size`，因为它们会生成批次。
- **verbose**: 日志显示模式，0 或 1。
- **steps**: 整数或 `None`。声明评估结束之前的总步数（批次样本）。默认值 `None` 时被忽略。
- **callbacks**: 一系列的 `keras.callbacks.Callback` 实例。一系列可以在预测时使用的回调函数。详见 [callbacks](#)。
- **max\_queue\_size**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。生成器队列的最大尺寸。若未指定，`max_queue_size` 将默认为 10。
- **workers**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。当使用基于进程的多线程时的最大进程数。若未指定，`workers` 将默认为 1。若为 0，将在主线程执行生成器。
- **use\_multiprocessing**: 布尔值。仅用于生成器或 `keras.utils.Sequence` 输入。如果是 `True`，使用基于进程的多线程。若未指定，`use_multiprocessing` 将默认为 `False`。注意由于这个实现依赖于 `multiprocessing`，你不应该像生成器传递不可选的参数，因为它们不能轻松地传递给子进程。

## 返回

预测的 Numpy 数组（或数组列表）。

## 异常

- **ValueError**: 在提供的输入数据与模型期望的不匹配的情况下，或者在有状态的模型接收到的样本不是 batch size 的倍数的情况下。

## train\_on\_batch

```
train_on_batch(x, y, sample_weight=None, class_weight=None,  
reset_metrics=True)
```

运行一批样品的单次梯度更新。

### 参数

- **x**: 训练数据的 Numpy 数组（如果模型只有一个输入），或者是 Numpy 数组的列表（如果模型有多个输入）。如果模型中的输入层被命名，你也可以传递一个字典，将输入层名称映射到 Numpy 数组。
- **y**: 目标（标签）数据的 Numpy 数组，或 Numpy 数组的列表（如果模型具有多个输出）。如果模型中的输出层被命名，你也可以传递一个字典，将输出层名称映射到 Numpy 数组。
- **sample\_weight**: 可选数组，与 x 长度相同，包含应用到模型损失函数的每个样本的权重。如果是时域数据，你可以传递一个尺寸为 (samples, sequence\_length) 的 2D 数组，为每一个样本的每一个时间步应用不同的权重。在这种情况下，你应该在 `compile()` 中指定 `sample_weight_mode="temporal"`。
- **class\_weight**: 可选的字典，用来映射类索引（整数）到权重（浮点）值，以在训练时对模型的损失函数加权。这可能有助于告诉模型「更多关注」来自代表性不足的类的样本。
- **reset\_metrics**: 如果为 `True`，返回的指标仅适用于该批次。如果为 `False`，则指标将在批次之间有状态地累积。

## 返回

标量训练误差（如果模型只有一个输入且没有评估标准），或者标量的列表（如果模型有多个输出和/或评估标准）。属性 `model.metrics_names` 将提供标量输出的显示标签。

## test\_on\_batch

```
test_on_batch(x, y, sample_weight=None, reset_metrics=True)
```

在一批样本上测试模型。

## 参数

- **x**: 测试数据的 Numpy 数组（如果模型只有一个输入），或者 Numpy 数组的列表（如果模型有多个输入）。如果模型中的输入层被命名，你也可以传递一个字典，将输入层名称映射到 Numpy 数组。
- **y**: 目标（标签）数据的 Numpy 数组，或 Numpy 数组的列表（如果模型具有多个输出）。如果模型中的输出层被命名，你也可以传递一个字典，将输出层名称映射到 Numpy 数组。
- **sample\_weight**: 可选数组，与 x 长度相同，包含应用到模型损失函数的每个样本的权重。如果是时域数据，你可以传递一个尺寸为 (samples, sequence\_length) 的 2D 数组，为每一个样本的每一个时间步应用不同的权重。
- **reset\_metrics**: 如果为 `True`，返回的指标仅适用于该批次。如果为 `False`，则指标将在批次之间有状态地累积。

## 返回

标量测试误差（如果模型只有一个输入且没有评估标准），或者标量的列表（如果模型有多个输出和/或评估标准）。属性 `model.metrics_names` 将提供标量输出的显示标签。

## predict\_on\_batch

```
predict_on_batch(x)
```

返回一批样本的模型预测值。

## 参数

- **x**: 输入数据，Numpy 数组。

## 返回

预测值的 Numpy 数组（或数组列表）。

## fit\_generator

```
fit_generator(generator, steps_per_epoch=None, epochs=1, verbose=1,
              callbacks=None, validation_data=None, validation_steps=None,
              validation_freq=1, class_weight=None, max_queue_size=10, workers=1,
              use_multiprocessing=False, shuffle=True, initial_epoch=0)
```

使用 Python 生成器（或 `Sequence` 实例）逐批生成的数据，按批次训练模型。

生成器与模型并行运行，以提高效率。例如，这可以让你在 CPU 上对图像进行实时数据增强，以在 GPU 上训练模型。

`keras.utils.Sequence` 的使用可以保证数据的顺序，以及当 `use_multiprocessing=True` 时，保证每个输入在每个 epoch 只使用一次。

## 参数

• **generator**: 一个生成器，或者一个 `Sequence` (`keras.utils.Sequence`) 对象的实例，以在使用多进程时避免数据的重复。生成器的输出应该为以下之一：

- 一个 `(inputs, targets)` 元组
- 一个 `(inputs, targets, sample_weights)` 元组。

这个元组（生成器的单个输出）组成了单个的 batch。因此，这个元组中的所有数组长度必须相同（与这一个 batch 的大小相等）。不同的 batch 可能大小不同。例如，一个 epoch 的最后一个 batch 往往比其他 batch 要小，如果数据集的尺寸不能被 batch size 整除。生成器将无限地在数据集上循环。当运行到第 `steps_per_epoch` 时，记一个 epoch 结束。

• **steps\_per\_epoch**: 在声明一个 epoch 完成并开始下一个 epoch 之前从 `generator` 产生的总步数（批次样本）。它通常应该等于 `ceil(num_samples / batch_size)`。对于 `Sequence`，它是可选的：如果未指定，将使用 `len(generator)` 作为步数。

• **epochs**: 整数。训练模型的迭代总轮数。一个 epoch 是对所提供的整个数据的一轮迭代，如 `steps_per_epoch` 所定义。注意，与 `initial_epoch` 一起使用，epoch 应被理解为「最后一轮」。模型没有经历由 `epochs` 给出的多次迭代的训练，而仅仅是直到达到索引 `epoch` 的轮次。

• **verbose**: 整数，0, 1 或 2。日志显示模式。0 = 安静模式，1 = 进度条，2 = 每轮一行。

• **callbacks**: 一系列的 `keras.callbacks.Callback` 实例。一系列可以在训练时使用的回调函数。详见 [callbacks](#)。

• **validation\_data**: 它可以是以下之一：

- 验证数据的生成器或 `Sequence` 实例
- `(inputs, targets)` 元组
- `(inputs, targets, sample_weights)` 元组。

在每个 epoch 结束时评估损失和任何模型指标。该模型不会对此数据进行训练。

• **validation\_steps**: 仅当 `validation_data` 是一个生成器时才可用。表示在每一轮迭代末尾停止前从 `validation_data` 生成器生成的总步数（样本批次）。它应该等于由 `batch_size` 分割的验证数据集的样本数。对于 `Sequence` 它是可选的：若未指定，将会使用 `len(validation_data)` 作为步数。

• **validation\_freq**: 只有在提供了验证数据时才有用。整数或 `collections.Container` 实例（例如列表、元组等）。如果是整数，指定在新的验证执行之前要执行多少次训练，例

如，`validation_freq=2` 在每 2 轮训练后执行验证。如果是 Container，指定执行验证的轮次，例如，`validation_freq=[1, 2, 10]` 表示在第 1、2、10 轮训练后执行验证。

- **class\_weight**: 可选的将类索引（整数）映射到权重（浮点）值的字典，用于加权损失函数（仅在训练期间）。这可以用来告诉模型「更多地关注」来自代表性不足的类的样本。
- **max\_queue\_size**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。生成器队列的最大尺寸。若未指定，`max_queue_size` 将默认为 10。
- **workers**: 整数。仅用于生成器或 `keras.utils.Sequence` 输入。当使用基于进程的多线程时的最大进程数。若未指定，`workers` 将默认为 1。若为 0，将在主线程执行生成器。
- **use\_multiprocessing**: 布尔值。仅用于生成器或 `keras.utils.Sequence` 输入。如果是 `True`，使用基于进程的多线程。若未指定，`use_multiprocessing` 将默认为 `False`。注意由于这个实现依赖于 `multiprocessing`，你不应该像生成器传递不可选的参数，因为它们不能轻松地传递给子进程。
- **shuffle**: 布尔值。是否在每轮迭代之前打乱 batch 的顺序。只能与 `Sequence` (`keras.utils.Sequence`) 实例同用。当 `steps_per_epoch` 为 `None` 是无效。
- **initial\_epoch**: 整数。开始训练的轮次（有助于恢复之前的训练）。

## 返回

一个 `History` 对象。其 `History.history` 属性是连续 epoch 训练损失和评估值，以及验证集损失和评估值的记录（如果适用）。

## 异常

- **ValueError**: 如果生成器生成的数据格式不正确。

## 示例

```
def generate_arrays_from_file(path):  
    while True:  
        with open(path) as f:  
            for line in f:  
                # 从文件中的每一行生成输入数据和标签的 numpy 数组,  
                x1, x2, y = process_line(line)  
                yield ({'input_1': x1, 'input_2': x2}, {'output': y})  
        f.close()  
  
model.fit_generator(generate_arrays_from_file('/my_file.txt'),  
                    steps_per_epoch=10000, epochs=10)
```

## evaluate\_generator

```
evaluate_generator(generator, steps=None, callbacks=None, max_queue_size=10,  
workers=1, use_multiprocessing=False, verbose=0)
```

在数据生成器上评估模型。

这个生成器应该返回与 `test_on_batch` 所接收的同样的数据。

## 参数

- **generator**: 一个生成 `(inputs, targets)` 或 `(inputs, targets, sample_weights)` 的生成器，或一个 `Sequence` (`keras.utils.Sequence`) 对象的实例，以避免在使用多进程时数据的重复。
- **steps**: 在声明一个 epoch 完成并开始下一个 epoch 之前从 `generator` 产生的总步数（批次样本）。它通常应该等于你的数据集的样本数量除以批量大小。对于 `Sequence`，它是可选的：如果未指定，将使用 `len(generator)` 作为步数。
- **callbacks**: 一系列的 `keras.callbacks.Callback` 实例。一系列可以在评估时使用的回调函数。详见 [callbacks](#)。
- **max\_queue\_size**: 生成器队列的最大尺寸。
- **workers**: 整数。使用的最大进程数量，如果使用基于进程的多线程。如未指定，`workers` 将默认为 1。如果为 0，将在主线程上执行生成器。
- **use\_multiprocessing**: 布尔值。如果 `True`，则使用基于进程的多线程。请注意，由于此实现依赖于多进程，所以不应将不可传递的参数传递给生成器，因为它们不能被轻易地传递给子进程。
- **verbose**: 日志显示模式，0 或 1。

## 返回

标量测试误差（如果模型只有一个输入且没有评估标准），或者标量的列表（如果模型有多个输出和/或评估标准）。属性 `model.metrics_names` 将提供标量输出的显示标签。

## 异常

- **ValueError**: 如果生成器生成的数据格式不正确。

## predict\_generator

```
predict_generator(generator, steps=None, callbacks=None, max_queue_size=10,  
workers=1, use_multiprocessing=False, verbose=0)
```

为来自数据生成器的输入样本生成预测。

这个生成器应该返回与 `predict_on_batch` 所接收的同样的数据。

## 参数

- **generator**: 生成器，返回批量输入样本，或一个 `Sequence` (`keras.utils.Sequence`) 对象的实例，以避免在使用多进程时数据的重复。
- **steps**: 在声明一个 epoch 完成并开始下一个 epoch 之前从 `generator` 产生的总步数（批次样本）。它通常应该等于你的数据集的样本数量除以批量大小。对于 `Sequence`，它是可选的：如果未指定，将使用 `len(generator)` 作为步数。
- **callbacks**: 一系列的 `keras.callbacks.Callback` 实例。一系列可以在预测时使用的回调函数。详见 [callbacks](#)。
- **max\_queue\_size**: 生成器队列的最大尺寸。
- **max\_queue\_size**: 生成器队列的最大尺寸。
- **workers**: 整数。使用的最大进程数量，如果使用基于进程的多线程。如未指定，`workers` 将默认为 1。如果为 0，将在主线程上执行生成器。
- **use\_multiprocessing**: 如果 True，则使用基于进程的多线程。请注意，由于此实现依赖于多进程，所以不应将不可传递的参数传递给生成器，因为它们不能被轻易地传递给子进程。
- **verbose**: 日志显示模式，0 或 1。

## 返回

预测值的 Numpy 数组（或数组列表）。

## 异常

- **ValueError**: 如果生成器生成的数据格式不正确。

## get\_layer

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

根据名称（唯一）或索引值查找网络层。

如果同时提供了 `name` 和 `index`，则 `index` 将优先。

索引值来自于水平图遍历的顺序（自下而上）。

## 参数

- **name**: 字符串，层的名字。
- **index**: 整数，层的索引。

## 返回

一个层实例。

## 异常

- **ValueError**: 如果层的名称或索引不正确。

# 关于 Keras 网络层

所有 Keras 网络层都有很多共同的函数：

- `layer.get_weights()`：以含有Numpy矩阵的列表形式返回层的权重。
- `layer.set_weights(weights)`：从含有Numpy矩阵的列表中设置层的权重（与 `get_weights` 的输出形状相同）。
- `layer.get_config()`：返回包含层配置的字典。此图层可以通过以下方式重置：

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

或：

```
from keras import layers

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

如果一个层具有单个节点 (i.e. 如果它不是共享层), 你可以得到它的输入张量、输出张量、输入尺寸和输出尺寸：

- `layer.input`
- `layer.output`
- `layer.input_shape`
- `layer.output_shape`

如果层有多个节点 (参见: [层节点和共享层的概念](#)), 您可以使用以下函数：

- `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)`



# 核心网络层

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)
```

就是你常用的全连接层。

Dense 实现以下操作：`output = activation(dot(input, kernel) + bias)` 其中 `activation` 是按逐个元素计算的激活函数，`kernel` 是由网络层创建的权值矩阵，以及 `bias` 是其创建的偏置向量（只在 `use_bias` 为 `True` 时才有用）。

- 注意：如果该层的输入的秩大于 2，那么它首先被展平然后再计算与 `kernel` 的点乘。

## 示例

```
# 作为 Sequential 模型的第一层
model = Sequential()
model.add(Dense(32, input_shape=(16,)))
# 现在模型就会以尺寸为 (*, 16) 的数组作为输入,
# 其输出数组的尺寸为 (*, 32)

# 在第一层之后, 你就不再需要指定输入的尺寸了:
model.add(Dense(32))
```

## 参数

- **units**: 正整数，输出空间维度。
- **activation**: 激活函数 (详见 [activations](#))。若不指定，则不使用激活函数 (即，线性激活:  
 $a(x) = x$ )。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: 运用到层的输出的正则化函数 (它的“activation”)。 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。

## 输入尺寸

nD 张量，尺寸: `(batch_size, ..., input_dim)`。最常见的情况是一个尺寸为 `(batch_size, input_dim)` 的 2D 输入。

## 输出尺寸

nD 张量，尺寸: `(batch_size, ..., units)`。例如，对于尺寸为 `(batch_size, input_dim)` 的 2D 输入，输出的尺寸为 `(batch_size, units)`。

## Activation

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

将激活函数应用于输出。

## 参数

- **activation:** 要使用的激活函数的名称 (详见: [activations](#))， 或者选择一个 Theano 或 TensorFlow 操作。

## 输入尺寸

任意尺寸。当使用此层作为模型中的第一层时， 使用参数 `input_shape` (整数元组，不包括样本数的轴) 。

## 输出尺寸

与输入相同。

Dropout

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

将 Dropout 应用于输入。

Dropout 包括在训练中每次更新时， 将输入单元的按比率随机设置为 0， 这有助于防止过拟合。

## 参数

- **rate**: 在 0 和 1 之间浮动。需要丢弃的输入比例。
- **noise\_shape**: 1D 整数张量， 表示将与输入相乘的二进制 dropout 掩层的形状。例如，如果你的输入尺寸为 `(batch_size, timesteps, features)`，然后你希望 dropout 掩层在所有时间步都是一样的，你可以使用 `noise_shape=(batch_size, 1, features)`。
- **seed**: 一个作为随机种子的 Python 整数。

## 参考文献

- [Dropout: A Simple Way to Prevent Neural Networks from Overfitting](#)

Flatten

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

将输入展平。不影响批量大小。

## 参数

- **data\_format**: 一个字符串，其值为 `channels_last`（默认值）或者 `channels_first`。它表明输入的维度的顺序。此参数的目的是当模型从一种数据格式切换到另一种数据格式时保留权重顺序。`channels_last` 对应着尺寸为 `(batch, ..., channels)` 的输入，而 `channels_first` 对应着尺寸为 `(batch, channels, ...)` 的输入。默认为 `image_data_format` 的值，你可以在 Keras 的配置文件 `~/.keras/keras.json` 中找到它。如果你从未设置过它，那么它将是 `channels_last`

## 示例

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

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



Input

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

`Input()` 用于实例化 Keras 张量。

Keras 张量是底层后端(Theano, TensorFlow 或 CNTK) 的张量对象，我们增加了一些特性，使得能够通过了解模型的输入 和输出来构建 Keras 模型。

例如，如果 `a`, `b` 和 `c` 都是 Keras 张量，那么以下操作是可行的：`model = Model(input=[a, b], output=c)`

添加的 Keras 属性是：

- `_keras_shape`：通过 Keras 端的尺寸推理 进行传播的整数尺寸元组。
- `_keras_history`：应用于张量的最后一层。整个网络层计算图可以递归地从该层中检索。

## 参数

- **shape**: 一个尺寸元组（整数），不包含批量大小。例如，`shape=(32,)` 表明期望的输入是按批次的 32 维向量。
- **batch\_shape**: 一个尺寸元组（整数），包含批量大小。例如，`batch_shape=(10, 32)` 表明期望的输入是 10 个 32 维向量。`batch_shape=(None, 32)` 表明任意批次大小的 32 维向量。
- **name**: 一个可选的层的名称的字符串。在一个模型中应该是唯一的（不可以重用一个名字两次）。如未提供，将自动生成。
- **dtype**: 输入所期望的数据类型，字符串表示 (`float32`, `float64`, `int32` ...)
- **sparse**: 一个布尔值，指明需要创建的占位符是否是稀疏的。
- **tensor**: 可选的可封装到 `Input` 层的现有张量。如果设定了，那么这个层将不会创建占位符张量。

## 返回

一个张量。

## 示例

```
# 这是 Keras 中的一个逻辑回归
x = Input(shape=(32,))
y = Dense(16, activation='softmax')(x)
model = Model(x, y)
```



Reshape

```
keras.layers.Reshape(target_shape)
```

将输入重新调整为特定的尺寸。

## 参数

- **target\_shape**: 目标尺寸。整数元组。不包含表示批量的轴。

## 输入尺寸

任意，尽管输入尺寸中的所有维度必须是固定的。当使用此层作为模型中的第一层时，使用参数 `input_shape`（整数元组，不包括样本数的轴）。

## 输出尺寸

```
(batch_size,) + target_shape
```

## 示例

```
# 作为 Sequential 模型的第一层
model = Sequential()
model.add(Reshape((3, 4), input_shape=(12,)))
# 现在: model.output_shape == (None, 3, 4)
# 注意: `None` 是批表示的维度

# 作为 Sequential 模型的中间层
model.add(Reshape((6, 2)))
# 现在: model.output_shape == (None, 6, 2)

# 还支持使用 `-1` 表示维度的尺寸推断
model.add(Reshape((-1, 2, 2)))
# 现在: model.output_shape == (None, 3, 2, 2)
```



Permute

```
keras.layers.Permute(dims)
```

根据给定的模式置换输入的维度。

在某些场景下很有用，例如将 RNN 和 CNN 连接在一起。

### 示例

```
model = Sequential()
model.add(Permute((2, 1), input_shape=(10, 64)))
# 现在: model.output_shape == (None, 64, 10)
# 注意: `None` 是批表示的维度
```

## 参数

- **dims**: 整数元组。置换模式，不包含样本维度。索引从 1 开始。例如, `(2, 1)` 置换输入的第一和第二个维度。

## 输入尺寸

任意。当使用此层作为模型中的第一层时，使用参数 `input_shape`（整数元组，不包括样本数的轴）。

## 输出尺寸

与输入尺寸相同，但是维度根据指定的模式重新排列。

RepeatVector

```
keras.layers.RepeatVector(n)
```

将输入重复 n 次。

示例

```
model = Sequential()
model.add(Dense(32, input_dim=32))
# 现在: model.output_shape == (None, 32)
# 注意: `None` 是批表示的维度

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

## 参数

- **n**: 整数，重复次数。

## 输入尺寸

2D 张量，尺寸为 `(num_samples, features)`。

## 输出尺寸

3D 张量，尺寸为 `(num_samples, n, features)`。

Lambda

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

将任意表达式封装为 `Layer` 对象。

### 示例

```
# 添加一个 x -> x^2 层
model.add(Lambda(x: x ** 2))
```



```
# 添加一个网络层，返回输入的正数部分
# 与负数部分的反面的连接

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))
```



```
# 添加一个返回 hadamard 乘积和两个输入张量之和的层

def hadamard_product_sum(tensors):
    out1 = tensors[0] * tensors[1]
    out2 = K.sum(out1, axis=-1)
    return [out1, out2]

def hadamard_product_sum_output_shape(input_shapes):
    shape1 = list(input_shapes[0])
    shape2 = list(input_shapes[1])
    assert shape1 == shape2 # 否则无法得到 hadamard 乘积
    return [tuple(shape1), tuple(shape2[:-1])]

x1 = Dense(32)(input_1)
x2 = Dense(32)(input_2)
layer = Lambda(hadamard_product_sum, hadamard_product_sum_output_shape)
x_hadamard, x_sum = layer([x1, x2])
```

## 参数

- **function**: 需要封装的函数。 将输入张量或张量序列作为第一个参数。
- **output\_shape**: 预期的函数输出尺寸。 只在使用 Theano 时有意义。 可以是元组或者函数。  
如果是元组，它只指定第一个维度； 样本维度假设与输入相同： `output_shape = (input_shape[0], ) + output_shape` 或者， 输入是 `None` 且样本维度也是 `None`：  
`output_shape = (None, ) + output_shape` 如果是函数，它指定整个尺寸为输入尺寸的一个函数： `output_shape = f(input_shape)`
- **mask**: 要么是 `None` (表示无 masking)， 要么是一个张量表示用于 Embedding 的输入 mask。
- **arguments**: 可选的需要传递给函数的关键字参数。

## 输入尺寸

任意。当使用此层作为模型中的第一层时， 使用参数 `input_shape` (整数元组， 不包括样本数的轴) 。

## 输出尺寸

由 `output_shape` 参数指定 (或者在使用 TensorFlow 时， 自动推理得到)。

ActivityRegularization

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

网络层，对基于代价函数的输入活动应用一个更新。

## 参数

- **l1**: L1 正则化因子 (正数浮点型)。
- **l2**: L2 正则化因子 (正数浮点型)。

## 输入尺寸

任意。当使用此层作为模型中的第一层时， 使用参数 `input_shape` (整数元组，不包括样本数的轴)。

## 输出尺寸

与输入相同。

Masking

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

使用覆盖值覆盖序列，以跳过时间步。

如果一个给定的样本时间步的所有特征都等于 `mask_value`，那么这个时间步将在所有下游层被覆盖（跳过）（只要它们支持覆盖）。

如果任何下游层不支持覆盖但仍然收到此类输入覆盖信息，会引发异常。

## 示例

考虑将要喂入一个 LSTM 层的 Numpy 矩阵 `x`，尺寸为 `(samples, timesteps, features)`。你想要覆盖时间步 #3 的样本 #0，以及时间步 #5 的样本 #2，由于你缺乏这几个时间步的特征。你可以：

- 设置 `x[0, 3, :] = 0.` 以及 `x[2, 5, :] = 0.`
- 在 LSTM 层之前，插入一个 `mask_value=0` 的 Masking 层：

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

## 参数

- **mask\_value**: 要么是 None, 要么是一个要跳过的 mask 值。

SpatialDropout1D

```
keras.layers.SpatialDropout1D(rate)
```

## Dropout 的 Spatial 1D 版本

此版本的功能与 Dropout 相同，但它会丢弃整个 1D 的特征图而不是丢弃单个元素。如果特征图中相邻的帧是强相关的（通常是靠前的卷积层中的情况），那么常规的 dropout 将无法使激活正则化，且导致有效的学习速率降低。在这种情况下，SpatialDropout1D 将有助于提高特征图之间的独立性，应该使用它来代替 Dropout。

### 参数

- **rate**: 0 到 1 之间的浮点数。需要丢弃的输入比例。

### 输入尺寸

3D 张量，尺寸为：`(samples, timesteps, channels)`

### 输出尺寸

与输入相同。

### 参考文献

- [Efficient Object Localization Using Convolutional Networks](#)

## SpatialDropout2D

```
keras.layers.SpatialDropout2D(rate, data_format=None)
```

### Dropout 的 Spatial 2D 版本

此版本的功能与 Dropout 相同，但它会丢弃整个 2D 的特征图而不是丢弃单个元素。如果特征图中相邻的像素是强相关的（通常是靠前的卷积层中的情况），那么常规的 dropout 将无法使激活正则化，且导致有效的学习速率降低。在这种情况下，SpatialDropout2D 将有助于提高特征图之间的独立性，应该使用它来代替 dropout。

## 参数

- **rate**: 0 到 1 之间的浮点数。需要丢弃的输入比例。
- **data\_format**: `channels_first` 或者 `channels_last`。在 `channels_first` 模式中，通道维度（即深度）位于索引 1，在 `channels_last` 模式中，通道维度位于索引 3。默认为 `image_data_format` 的值，你可以在 Keras 的配置文件 `~/.keras/keras.json` 中找到它。如果你从未设置过它，那么它将是 `channels_last`。

## 输入尺寸

4D 张量，如果 `data_format=channels_first`，尺寸为 `(samples, channels, rows, cols)`，如果 `data_format=channels_last`，尺寸为 `(samples, rows, cols, channels)`。

## 输出尺寸

与输入相同。

## 参考文献

- [Efficient Object Localization Using Convolutional Networks](#)

## SpatialDropout3D

[\[source\]](#)

```
keras.layers.SpatialDropout3D(rate, data_format=None)
```

Dropout 的 Spatial 3D 版本

此版本的功能与 Dropout 相同，但它会丢弃整个 3D 的特征图而不是丢弃单个元素。如果特征图中相邻的体素是强相关的（通常是靠前的卷积层中的情况），那么常规的 dropout 将无法使激活正则化，且导致有效的学习速率降低。在这种情况下，SpatialDropout3D 将有助于提高特征图之间的独立性，应该使用它来代替 dropout。

## 参数

- **rate**: 0 到 1 之间的浮点数。需要丢弃的输入比例。
- **data\_format**: `channels_first` 或者 `channels_last`。在 `channels_first` 模式中，通道维度（即深度）位于索引 1，在 `channels_last` 模式中，通道维度位于索引 4。默认为 `image_data_format` 的值，你可以在 Keras 的配置文件 `~/.keras/keras.json` 中找到它。如果你从未设置过它，那么它将是 `channels_last`。

## 输入尺寸

5D 张量，如果 `data_format=channels_first`，尺寸为 `(samples, channels, dim1, dim2, dim3)`，如果 `data_format=channels_last`，尺寸为 `(samples, dim1, dim2, dim3, channels)`。

## 输出尺寸

与输入相同。

## 参考文献

- [Efficient Object Localization Using Convolutional Networks](#)



# 卷积层 Convolutional Layers

Conv1D

```
keras.layers.Conv1D(filters, kernel_size, strides=1, padding='valid',
data_format='channels_last', 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 卷积层 (例如时序卷积)。

该层创建了一个卷积核，该卷积核以单个空间（或时间）维上的层输入进行卷积，以生成输出张量。如果 `use_bias` 为 `True`，则会创建一个偏置向量并将其添加到输出中。最后，如果 `activation` 不是 `None`，它也会应用于输出。

当使用该层作为模型第一层时，需要提供 `input_shape` 参数（整数元组或 `None`，不包含 `batch` 轴），例如，`input_shape=(10, 128)` 在 `data_format="channels_last"` 时表示 10 个 128 维的向量组成的向量序列，`(None, 128)` 表示每步 128 维的向量组成的变长序列。

## 参数

- **`filters`**: 整数，输出空间的维度（即卷积中滤波器的输出数量）。
- **`kernel_size`**: 一个整数，或者单个整数表示的元组或列表，指明 1D 卷积窗口的长度。
- **`strides`**: 一个整数，或者单个整数表示的元组或列表，指明卷积的步长。指定任何 `stride` 值  $\neq 1$  与指定 `dilation_rate` 值  $\neq 1$  两者不兼容。
- **`padding`**: `"valid"`, `"causal"` 或 `"same"` 之一 (大小写敏感) `"valid"` 表示「不填充」。  
`"same"` 表示填充输入以使输出具有与原始输入相同的长度。`"causal"` 表示因果（膨胀）卷积，例如，`output[t]` 不依赖于 `input[t+1:]`，在模型不应违反时间顺序的时间数据建模时非常有用。详见 [WaveNet: A Generative Model for Raw Audio, section 2.1](#)。
- **`data_format`**: 字符串，`"channels_last"` (默认) 或 `"channels_first"` 之一。输入的各个维度顺序。`"channels_last"` 对应输入尺寸为 `(batch, steps, channels)` (Keras 中时序数据的默认格式) 而 `"channels_first"` 对应输入尺寸为 `(batch, channels, steps)`。
- **`dilation_rate`**: 一个整数，或者单个整数表示的元组或列表，指定用于膨胀卷积的膨胀率。  
当前，指定任何 `dilation_rate` 值  $\neq 1$  与指定 `stride` 值  $\neq 1$  两者不兼容。
- **`activation`**: 要使用的激活函数 (详见 [activations](#))。如未指定，则不使用激活函数 (即线性激活：`a(x) = x`)。
- **`use_bias`**: 布尔值，该层是否使用偏置向量。
- **`kernel_initializer`**: `kernel` 权值矩阵的初始化器 (详见 [initializers](#))。
- **`bias_initializer`**: 偏置向量的初始化器 (详见 [initializers](#))。
- **`kernel_regularizer`**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **`bias_regularizer`**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **`activity_regularizer`**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。
- **`kernel_constraint`**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **`bias_constraint`**: 运用到偏置向量的约束函数 (详见 [constraints](#))。

## 输入尺寸

3D 张量，尺寸为 `(batch_size, steps, input_dim)`。

## 输出尺寸

3D 张量，尺寸为 `(batch_size, new_steps, filters)`。由于填充或窗口按步长滑动，`steps` 值可能已更改。

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 卷积层 (例如对图像的空间卷积)。

该层创建了一个卷积核，该卷积核对层输入进行卷积，以生成输出张量。如果 `use_bias` 为 `True`，则会创建一个偏置向量并将其添加到输出中。最后，如果 `activation` 不是 `None`，它也会应用于输出。

当使用该层作为模型第一层时，需要提供 `input_shape` 参数（整数元组，不包含 batch 轴），例如，`input_shape=(128, 128, 3)` 表示 128x128 RGB 图像，在 `data_format="channels_last"` 时。

## 参数

- **`filters`**: 整数，输出空间的维度（即卷积中滤波器的输出数量）。
- **`kernel_size`**: 一个整数，或者 2 个整数表示的元组或列表，指明 2D 卷积窗口的宽度和高度。可以是一个整数，为所有空间维度指定相同的值。
- **`strides`**: 一个整数，或者 2 个整数表示的元组或列表，指明卷积沿宽度和高度方向的步长。可以是一个整数，为所有空间维度指定相同的值。指定任何 `stride` 值 != 1 与指定 `dilation_rate` 值 != 1 两者不兼容。
- **`padding`**: `"valid"` 或 `"same"` (大小写敏感)。
- **`data_format`**: 字符串，`channels_last` (默认) 或 `channels_first` 之一，表示输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, height, width, channels)`，`channels_first` 对应输入尺寸为 `(batch, channels, height, width)`。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用 `channels_last`。
- **`dilation_rate`**: 一个整数或 2 个整数的元组或列表，指定膨胀卷积的膨胀率。可以是一个整数，为所有空间维度指定相同的值。当前，指定任何 `dilation_rate` 值 != 1 与指定 `stride` 值 != 1 两者不兼容。
- **`activation`**: 要使用的激活函数 (详见 [activations](#))。如果你不指定，则不使用激活函数 (即线性激活： $a(x) = x$ )。
- **`use_bias`**: 布尔值，该层是否使用偏置向量。
- **`kernel_initializer`**: `kernel` 权值矩阵的初始化器 (详见 [initializers](#))。
- **`bias_initializer`**: 偏置向量的初始化器 (详见 [initializers](#))。
- **`kernel_regularizer`**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **`bias_regularizer`**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **`activity_regularizer`**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。
- **`kernel_constraint`**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **`bias_constraint`**: 运用到偏置向量的约束函数 (详见 [constraints](#))。

## 输入尺寸

- 如果 `data_format='channels_first'`, 输入 4D 张量, 尺寸为 `(samples, channels, rows, cols)`。
- 如果 `data_format='channels_last'`, 输入 4D 张量, 尺寸为 `(samples, rows, cols, channels)`。

## 输出尺寸

- 如果 `data_format='channels_first'`, 输出 4D 张量, 尺寸为 `(samples, filters, new_rows, new_cols)`。
- 如果 `data_format='channels_last'`, 输出 4D 张量, 尺寸为 `(samples, new_rows, new_cols, filters)`。

由于填充的原因, `rows` 和 `cols` 值可能已更改。

SeparableConv1D

```
keras.layers.SeparableConv1D(filters, kernel_size, strides=1, padding='valid',
data_format='channels_last', dilation_rate=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)
```

深度方向的可分离 1D 卷积。

可分离的卷积的操作包括，首先执行深度方向的空间卷积（分别作用于每个输入通道），紧接着一个将所得输出通道混合在一起的逐点卷积。`depth_multiplier` 参数控制深度步骤中每个输入通道生成多少个输出通道。

直观地说，可分离的卷积可以理解为一种将卷积核分解成两个较小的卷积核的方法，或者作为 Inception 块的一个极端版本。

## 参数

- **filters**: 整数，输出空间的维度（即卷积中滤波器的输出数量）。
- **kernel\_size**: 一个整数，或者单个整数表示的元组或列表，指明 1D 卷积窗口的长度。
- **strides**: 一个整数，或者单个整数表示的元组或列表，指明卷积的步长。指定任何 `stride` 值  $\neq 1$  与指定 `dilation_rate` 值  $\neq 1$  两者不兼容。
- **padding**: `"valid"` 或 `"same"` (大小写敏感)。
- **data\_format**: 字符串，`channels_last` (默认) 或 `channels_first` 之一，表示输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, height, width, channels)`，`channels_first` 对应输入尺寸为 `(batch, channels, height, width)`。它默认从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用「`channels_last`」。
- **dilation\_rate**: 一个整数，或者单个整数表示的元组或列表，为使用扩张（空洞）卷积指明扩张率。目前，指定任何 `dilation_rate` 值  $\neq 1$  与指定任何 `stride` 值  $\neq 1$  两者不兼容。
- **depth\_multiplier**: 每个输入通道的深度方向卷积输出通道的数量。深度方向卷积输出通道的总数将等于 `filters_in * depth_multiplier`。
- **activation**: 要使用的激活函数 (详见 [activations](#))。如果你不指定，则不使用激活函数 (即线性激活: `a(x) = x`)。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **depthwise\_initializer**: 运用到深度方向的核矩阵的初始化器 (详见 [initializers](#))。
- **pointwise\_initializer**: 运用到逐点核矩阵的初始化器 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **depthwise\_regularizer**: 运用到深度方向的核矩阵的正则化函数 (详见 [regularizer](#))。
- **pointwise\_regularizer**: 运用到逐点核矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: 运用到层输出（它的激活值）的正则化函数 (详见 [regularizer](#))。
- **depthwise\_constraint**: 运用到深度方向的核矩阵的约束函数 (详见 [constraints](#))。
- **pointwise\_constraint**: 运用到逐点核矩阵的约束函数 (详见 [constraints](#))。

- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。

## 输入尺寸

- 如果 `data_format='channels_first'`, 输入 3D 张量, 尺寸为 `(batch, channels, steps)`。
- 如果 `data_format='channels_last'`, 输入 3D 张量, 尺寸为 `(batch, steps, channels)`。

## 输出尺寸

- 如果 `data_format='channels_first'`, 输出 3D 张量, 尺寸为  
`(batch, filters, new_steps)`。
- 如果 `data_format='channels_last'`, 输出 3D 张量, 尺寸为 `(batch, new_steps, filters)`
  -

由于填充的原因, `new_steps` 值可能已更改。

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)
```

深度方向的可分离 2D 卷积。

可分离的卷积的操作首先执行深度方向的空间卷积（分别作用于每个输入通道），紧接一个将所得输出通道混合在一起的逐点卷积。`depth_multiplier` 参数控制深度步骤中每个输入通道生成多少个输出通道。

直观地说，可分离的卷积可以理解为一种将卷积核分解成两个较小的卷积核的方法，或者作为 Inception 块的一个极端版本。

## 参数

- **filters**: 整数，输出空间的维度（即卷积中滤波器的输出数量）。
- **kernel\_size**: 一个整数，或者 2 个整数表示的元组或列表，指明 2D 卷积窗口的高度和宽度。可以是一个整数，为所有空间维度指定相同的值。
- **strides**: 一个整数，或者 2 个整数表示的元组或列表，指明卷积沿高度和宽度方向的步长。可以是一个整数，为所有空间维度指定相同的值。指定任何 `stride` 值 != 1 与指定 `dilation_rate` 值 != 1 两者不兼容。
- **padding**: "valid" 或 "same" (大小写敏感)。
- **data\_format**: 字符串，`channels_last` (默认) 或 `channels_first` 之一，表示输入中维度的顺序。`channels_last` 对应输入尺寸为 (`batch, height, width, channels`)，`channels_first` 对应输入尺寸为 (`batch, channels, height, width`)。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用「`channels_last`」。
- **dilation\_rate**: 一个整数，或者 2 个整数表示的元组或列表，为使用扩张（空洞）卷积指明扩张率。目前，指定任何 `dilation_rate` 值 != 1 与指定任何 `stride` 值 != 1 两者不兼容。
- **depth\_multiplier**: 每个输入通道的深度方向卷积输出通道的数量。深度方向卷积输出通道的总数将等于 `filters_in * depth_multiplier`。
- **activation**: 要使用的激活函数 (详见 [activations](#))。如果你不指定，则不使用激活函数 (即线性激活: `a(x) = x`)。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **depthwise\_initializer**: 运用到深度方向的核矩阵的初始化器 (详见 [initializers](#))。
- **pointwise\_initializer**: 运用到逐点核矩阵的初始化器 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **depthwise\_regularizer**: 运用到深度方向的核矩阵的正则化函数 (详见 [regularizer](#))。
- **pointwise\_regularizer**: 运用到逐点核矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。

- **depthwise\_constraint**: 运用到深度方向的核矩阵的约束函数 (详见 [constraints](#))。
- **pointwise\_constraint**: 运用到逐点核矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。

## 输入尺寸

- 如果 `data_format='channels_first'`, 输入 4D 张量, 尺寸为 `(batch, channels, rows, cols)`。
- 如果 `data_format='channels_last'`, 输入 4D 张量, 尺寸为 `(batch, rows, cols, channels)`。

## 输出尺寸

- 如果 `data_format='channels_first'`, 输出 4D 张量, 尺寸为 `(batch, filters, new_rows, new_cols)`。
- 如果 `data_format='channels_last'`, 输出 4D 张量, 尺寸为 `(batch, new_rows, new_cols, filters)`。

由于填充的原因, `rows` 和 `cols` 值可能已更改。

DepthwiseConv2D

```
keras.layers.DepthwiseConv2D(kernel_size, strides=(1, 1), padding='valid',
depth_multiplier=1, data_format=None, dilation_rate=(1, 1), activation=None,
use_bias=True, depthwise_initializer='glorot_uniform',
bias_initializer='zeros', depthwise_regularizer=None, bias_regularizer=None,
activity_regularizer=None, depthwise_constraint=None, bias_constraint=None)
```

深度 2D 卷积。

深度卷积仅执行深度空间卷积中的第一步（其分别作用于每个输入通道）。`depth_multiplier` 参数控制深度步骤中每个输入通道生成多少个输出通道。

## Arguments

- **kernel\_size**: 一个整数，或者 2 个整数表示的元组或列表，指明 2D 卷积窗口的高度和宽度。可以是一个整数，为所有空间维度指定相同的值。
- **strides**: 一个整数，或者 2 个整数表示的元组或列表，指明卷积沿高度和宽度方向的步长。可以是一个整数，为所有空间维度指定相同的值。指定任何 `stride` 值 != 1 与指定 `dilation_rate` 值 != 1 两者不兼容。
- **padding**: "valid" 或 "same" (大小写敏感)。
- **depth\_multiplier**: 每个输入通道的深度方向卷积输出通道的数量。深度方向卷积输出通道的总数将等于 `filters_in * depth_multiplier`。
- **data\_format**: 字符串，`channels_last` (默认) 或 `channels_first` 之一，表示输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, height, width, channels)`，`channels_first` 对应输入尺寸为 `(batch, channels, height, width)`。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用「`channels_last`」。
- **dilation\_rate**: 一个整数，或者 2 个整数表示的元组或列表，为使用扩张（空洞）卷积指明扩张率。目前，指定任何 `dilation_rate` 值 != 1 与指定任何 `stride` 值 != 1 两者不兼容。
- **activation**: 要使用的激活函数 (详见 [activations](#))。如果你不指定，则不使用激活函数 (即线性激活： $a(x) = x$ )。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **depthwise\_initializer**: 运用到深度方向的核矩阵的初始化器 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **depthwise\_regularizer**: 运用到深度方向的核矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。
- **depthwise\_constraint**: 运用到深度方向的核矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。

## 输入尺寸

- 如果 `data_format='channels_first'`，输入 4D 张量，尺寸为 `(batch, channels, rows, cols)`。

- 如果 `data_format='channels_last'`, 输入 4D 张量, 尺寸为 `(batch, rows, cols, channels)`。

## 输出尺寸

- 如果 `data_format='channels_first'`, 输出 4D 张量, 尺寸为 `(batch, channels * depth_multiplier, new_rows, new_cols)`。
- 如果 `data_format='channels_last'`, 输出 4D 张量, 尺寸为 `(batch, new_rows, new_cols, channels * depth_multiplier)`。

由于填充的原因, `rows` 和 `cols` 值可能已更改。

Conv2DTranspose

```
keras.layers.Conv2DTranspose(filters, kernel_size, strides=(1, 1),
padding='valid', output_padding=None, 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)
```

转置卷积层 (有时被成为反卷积)。

对转置卷积的需求一般来自希望使用与正常卷积相反方向的变换，即，将具有卷积输出尺寸的东西转换为具有卷积输入尺寸的东西，同时保持与所述卷积相容的连通性模式。

当使用该层作为模型第一层时，需要提供 `input_shape` 参数（整数元组，不包含 batch 轴），例如，`input_shape=(128, 128, 3)` 表示 128x128 RGB 图像，在 `data_format="channels_last"` 时。

## 参数

- **filters**: 整数，输出空间的维度（即卷积中滤波器的输出数量）。
- **kernel\_size**: 一个整数，或者 2 个整数表示的元组或列表，指明 2D 卷积窗口的高度和宽度。可以是一个整数，为所有空间维度指定相同的值。
- **strides**: 一个整数，或者 2 个整数表示的元组或列表，指明卷积沿高度和宽度方向的步长。可以是一个整数，为所有空间维度指定相同的值。指定任何 `stride` 值  $\neq 1$  与指定 `dilation_rate` 值  $\neq 1$  两者不兼容。
- **padding**: `"valid"` 或 `"same"` (大小写敏感)。
- **output\_padding**: 一个整数，或者 2 个整数表示的元组或列表，指定沿输出张量的高度和宽度的填充量。可以是单个整数，以指定所有空间维度的相同值。沿给定维度的输出填充量必须低于沿同一维度的步长。如果设置为 `None` (默认)，输出尺寸将自动推理出来。
- **data\_format**: 字符串，`channels_last` (默认) 或 `channels_first` 之一，表示输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, height, width, channels)`，`channels_first` 对应输入尺寸为 `(batch, channels, height, width)`。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用“`channels_last`”。
- **dilation\_rate**: 一个整数或 2 个整数的元组或列表，指定膨胀卷积的膨胀率。可以是一个整数，为所有空间维度指定相同的值。当前，指定任何 `dilation_rate` 值  $\neq 1$  与指定 `stride` 值  $\neq 1$  两者不兼容。
- **activation**: 要使用的激活函数 (详见 [activations](#))。如果你不指定，则不使用激活函数 (即线性激活：`a(x) = x`)。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。

- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。

## 输入尺寸

- 如果 `data_format='channels_first'`, 输入 4D 张量, 尺寸为 `(batch, channels, rows, cols)`。
- 如果 `data_format='channels_last'`, 输入 4D 张量, 尺寸为 `(batch, rows, cols, channels)`。

## 输出尺寸

- 如果 `data_format='channels_first'`, 输出 4D 张量, 尺寸为 `(batch, filters, new_rows, new_cols)`。
- 如果 `data_format='channels_last'`, 输出 4D 张量, 尺寸为 `(batch, new_rows, new_cols, filters)`。

由于填充的原因, `rows` 和 `cols` 值可能已更改。

如果指定了 `output_padding`:

```
new_rows = ((rows - 1) * strides[0] + kernel_size[0]
            - 2 * padding[0] + output_padding[0])
new_cols = ((cols - 1) * strides[1] + kernel_size[1]
            - 2 * padding[1] + output_padding[1])
```

## 参考文献

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

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 卷积层 (例如立体空间卷积)。

该层创建了一个卷积核，该卷积核对层输入进行卷积，以生成输出张量。如果 `use_bias` 为 `True`，则会创建一个偏置向量并将其添加到输出中。最后，如果 `activation` 不是 `None`，它也会应用于输出。

当使用该层作为模型第一层时，需要提供 `input_shape` 参数（整数元组，不包含 batch 轴），例如，`input_shape=(128, 128, 128, 1)` 表示  $128 \times 128 \times 128$  的单通道立体，在 `data_format="channels_last"` 时。

## 参数

- **`filters`**: 整数，输出空间的维度（即卷积中滤波器的输出数量）。
- **`kernel_size`**: 一个整数，或者 3 个整数表示的元组或列表，指明 3D 卷积窗口的深度、高度和宽度。可以是一个整数，为所有空间维度指定相同的值。
- **`strides`**: 一个整数，或者 3 个整数表示的元组或列表，指明卷积沿每一个空间维度的步长。可以是一个整数，为所有空间维度指定相同的步长值。指定任何 `stride` 值  $\neq 1$  与指定 `dilation_rate` 值  $\neq 1$  两者不兼容。
- **`padding`**: `"valid"` 或 `"same"` (大小写敏感)。
- **`data_format`**: 字符串，`channels_last` (默认) 或 `channels_first` 之一，表示输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)`，`channels_first` 对应输入尺寸为 `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用 `"channels_last"`。
- **`dilation_rate`**: 一个整数或 3 个整数的元组或列表，指定膨胀卷积的膨胀率。可以是一个整数，为所有空间维度指定相同的值。当前，指定任何 `dilation_rate` 值  $\neq 1$  与指定 `stride` 值  $\neq 1$  两者不兼容。
- **`activation`**: 要使用的激活函数 (详见 [activations](#))。如果你不指定，则不使用激活函数 (即线性激活： $a(x) = x$ )。
- **`use_bias`**: 布尔值，该层是否使用偏置向量。
- **`kernel_initializer`**: `kernel` 权值矩阵的初始化器 (详见 [initializers](#))。
- **`bias_initializer`**: 偏置向量的初始化器 (详见 [initializers](#))。
- **`kernel_regularizer`**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **`bias_regularizer`**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **`activity_regularizer`**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。
- **`kernel_constraint`**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **`bias_constraint`**: 运用到偏置向量的约束函数 (详见 [constraints](#))。

## 输入尺寸

- 如果 `data_format='channels_first'`, 输入 5D 张量, 尺寸为 `(samples, channels, conv_dim1, conv_dim2, conv_dim3)`。
- 如果 `data_format='channels_last'`, 输入 5D 张量, 尺寸为 `(samples, conv_dim1, conv_dim2, conv_dim3, channels)`。

## 输出尺寸

- 如果 `data_format='channels_first'`, 输出 5D 张量, 尺寸为 `(samples, filters, new_conv_dim1, new_conv_dim2, new_conv_dim3)`。
- 如果 `data_format='channels_last'`, 输出 5D 张量, 尺寸为 `(samples, new_conv_dim1, new_conv_dim2, new_conv_dim3, filters)`。

由于填充的原因, `new_conv_dim1`, `new_conv_dim2` 和 `new_conv_dim3` 值可能已更改。

Conv3DTranspose

```
keras.layers.Conv3DTranspose(filters, kernel_size, strides=(1, 1, 1),  
padding='valid', output_padding=None, 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)
```

转置卷积层 (有时被成为反卷积)。

对转置卷积的需求一般来自希望使用与正常卷积相反方向的变换，即，将具有卷积输出尺寸的东西转换为具有卷积输入尺寸的东西，同时保持与所述卷积相容的连通性模式。

当使用该层作为模型第一层时，需要提供 `input_shape` 参数（整数元组，不包含样本表示的轴），例如，`input_shape=(128, 128, 128, 3)` 表示尺寸 128x128x128 的 3 通道立体，在 `data_format="channels_last"` 时。

## 参数

- **filters**: 整数，输出空间的维度（即卷积中滤波器的输出数量）。
- **kernel\_size**: 一个整数，或者 3 个整数表示的元组或列表，指明 3D 卷积窗口的深度、高度和宽度。可以是一个整数，为所有空间维度指定相同的值。
- **strides**: 一个整数，或者 3 个整数表示的元组或列表，指明沿深度、高度和宽度方向的步长。可以是一个整数，为所有空间维度指定相同的值。指定任何 `stride` 值 != 1 与指定 `dilation_rate` 值 != 1 两者不兼容。
- **padding**: "valid" 或 "same" (大小写敏感)。
- **output\_padding**: 一个整数，或者 3 个整数表示的元组或列表，指定沿输出张量的高度和宽度的填充量。可以是单个整数，以指定所有空间维度的相同值。沿给定维度的输出填充量必须低于沿同一维度的步长。如果设置为 `None` (默认)，输出尺寸将自动推理出来。
- **data\_format**: 字符串，`channels_last` (默认) 或 `channels_first` 之一，表示输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, depth, height, width, channels)`，`channels_first` 对应输入尺寸为 `(batch, channels, depth, height, width)`。它默认认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用「`channels_last`」。
- **dilation\_rate**: 一个整数或 3 个整数的元组或列表，指定膨胀卷积的膨胀率。可以是一个整数，为所有空间维度指定相同的值。当前，指定任何 `dilation_rate` 值 != 1 与指定 `stride` 值 != 1 两者不兼容。
- **activation**: 要使用的激活函数 (详见 [activations](#))。如果你不指定，则不使用激活函数 (即线性激活：`a(x) = x`)。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。

- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。

## 输入尺寸

如果 `data_format='channels_first'`, 输入 5D 张量, 尺寸为  
`(batch, channels, depth, rows, cols)`, 如果 `data_format='channels_last'`, 输入 5D 张量, 尺寸为 `(batch, depth, rows, cols, channels)`。

## Output shape

如果 `data_format='channels_first'`, 输出 5D 张量, 尺寸为 `(batch, filters, new_depth, new_rows, new_cols)`, 如果 `data_format='channels_last'`, 输出 5D 张量, 尺寸为 `(batch, new_depth, new_rows, new_cols, filters)`。

`depth` 和 `rows` 和 `cols` 可能因为填充而改变。如果指定了 `output_padding`:

```
new_depth = ((depth - 1) * strides[0] + kernel_size[0]
             - 2 * padding[0] + output_padding[0])
new_rows = ((rows - 1) * strides[1] + kernel_size[1]
            - 2 * padding[1] + output_padding[1]))
new_cols = ((cols - 1) * strides[2] + kernel_size[2]
            - 2 * padding[2] + output_padding[2])
```

## 参考文献

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

Cropping1D

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

1D 输入的裁剪层（例如时间序列）。

它沿着时间维度（第 1 个轴）裁剪。

## 参数

- **cropping**: 整数或整数元组（长度为 2）。在裁剪维度（第 1 个轴）的开始和结束位置应该裁剪多少个单位。如果只提供了一个整数，那么这两个位置将使用相同的值。

## 输入尺寸

3D 张量，尺寸为 `(batch, axis_to_crop, features)`。

## 输出尺寸

3D 张量，尺寸为 `(batch, cropped_axis, features)`。

Cropping2D

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

2D 输入的裁剪层（例如图像）。

它沿着空间维度裁剪，即宽度和高度。

## 参数

- **cropping**: 整数，或 2 个整数的元组，或 2 个整数的 2 个元组。

- 如果为整数：将对宽度和高度应用相同的对称裁剪。

- 如果为 2 个整数的元组：解释为对高度和宽度的两个不同的对称裁剪值：

- `(symmetric_height_crop, symmetric_width_crop)`。

- 如果为 2 个整数的 2 个元组：解释为 `((top_crop, bottom_crop), (left_crop, right_crop))`。

- **data\_format**: 字符串，`channels_last` (默认) 或 `channels_first` 之一，表示输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, height, width, channels)`，`channels_first` 对应输入尺寸为 `(batch, channels, height, width)`。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用“`channels_last`”。

## 输出尺寸

- 如果 `data_format='channels_last'`，输出 4D 张量，尺寸为 `(batch, rows, cols, channels)`。

- 如果 `data_format='channels_first'`，输出 4D 张量，尺寸为 `(batch, channels, rows, cols)`。

由于填充的原因，`rows` 和 `cols` 值可能已更改。

## 输入尺寸

- 如果 `data_format` 为 “`channels_last`”，输入 4D 张量，尺寸为 `(batch, cropped_rows, cropped_cols, channels)`。

- 如果 `data_format` 为 “`channels_first`”，输入 4D 张量，尺寸为 `(batch, channels, cropped_rows, cropped_cols)`。

## 输出尺寸

- 如果 `data_format` 为 “`channels_last`”，输出 4D 张量，尺寸为 `(batch, cropped_rows, cropped_cols, channels)`

- 如果 `data_format` 为 “`channels_first`”，输出 4D 张量，尺寸为 `(batch, channels, cropped_rows, cropped_cols)`。

## 示例

```
# 裁剪输入的 2D 图像或特征图
model = Sequential()
model.add(Cropping2D(cropping=((2, 2), (4, 4)),
                     input_shape=(28, 28, 3)))
# 现在 model.output_shape == (None, 24, 20, 3)
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Cropping2D(cropping=((2, 2), (2, 2))))
# 现在 model.output_shape == (None, 20, 16, 64)
```



Cropping3D

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

3D 数据的裁剪层（例如空间或时空）。

## 参数

- **cropping**: 整数，或 3 个整数的元组，或 2 个整数的 3 个元组。
  - 如果为整数： 将对深度、高度和宽度应用相同的对称裁剪。
  - 如果为 3 个整数的元组： 解释为对深度、高度和高度的 3 个不同的对称裁剪值：  
`(symmetric_dim1_crop, symmetric_dim2_crop, symmetric_dim3_crop)`。
  - 如果为 2 个整数的 3 个元组： 解释为 `((left_dim1_crop, right_dim1_crop), (left_dim2_crop, right_dim2_crop), (left_dim3_crop, right_dim3_crop))`。
- **data\_format**: 字符串，`channels_last` (默认) 或 `channels_first` 之一，表示输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)`，`channels_first` 对应输入尺寸为 `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用“`channels_last`”。

## 输入尺寸

5D 张量，尺寸为：

- 如果 `data_format` 为 "`channels_last`"：  
`(batch, first_cropped_axis, second_cropped_axis, third_cropped_axis, depth)`
- 如果 `data_format` 为 "`channels_first`"：  
`(batch, depth, first_cropped_axis, second_cropped_axis, third_cropped_axis)`

## 输出尺寸

5D 张量，尺寸为：

- 如果 `data_format` 为 "`channels_last`"：  
`(batch, first_cropped_axis, second_cropped_axis, third_cropped_axis, depth)`
- 如果 `data_format` 为 "`channels_first`"：  
`(batch, depth, first_cropped_axis, second_cropped_axis, third_cropped_axis)`。

UpSampling1D

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

1D 输入的上采样层。

沿着时间轴重复每个时间步 `size` 次。

## 参数

- **size**: 整数。上采样因子。

## 输入尺寸

3D 张量，尺寸为 `(batch, steps, features)`。

## 输出尺寸

3D 张量，尺寸为 `(batch, upsampled_steps, features)`。

UpSampling2D

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

2D 输入的上采样层。

沿着数据的行和列分别重复 `size[0]` 和 `size[1]` 次。

## 参数

- **size**: 整数，或 2 个整数的元组。行和列的上采样因子。
- **data\_format**: 字符串，`channels_last` (默认) 或 `channels_first` 之一，表示输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, height, width, channels)`，`channels_first` 对应输入尺寸为 `(batch, channels, height, width)`。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用 “`channels_last`”。
- **interpolation**: 字符串，`nearest` 或 `bilinear` 之一。注意 CNTK 暂不支持 `bilinear upscaling`，以及对于 Theano，只可以使用 `size=(2, 2)`。

## 输入尺寸

- 如果 `data_format` 为 `"channels_last"`，输入 4D 张量，尺寸为 `(batch, rows, cols, channels)`。
- 如果 `data_format` 为 `"channels_first"`，输入 4D 张量，尺寸为 `(batch, channels, rows, cols)`。

## 输出尺寸

- 如果 `data_format` 为 `"channels_last"`，输出 4D 张量，尺寸为 `(batch, upsampled_rows, upsampled_cols, channels)`。
- 如果 `data_format` 为 `"channels_first"`，输出 4D 张量，尺寸为 `(batch, channels, upsampled_rows, upsampled_cols)`。

UpSampling3D

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

3D 输入的上采样层。

沿着数据的第 1、2、3 维度分别重复 `size[0]`、`size[1]` 和 `size[2]` 次。

## 参数

- **size**: 整数，或 3 个整数的元组。`dim1`, `dim2` 和 `dim3` 的上采样因子。
- **data\_format**: 字符串，`channels_last` (默认) 或 `channels_first` 之一，表示输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)`，`channels_first` 对应输入尺寸为 `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用“`channels_last`”。

## 输入尺寸

- 如果 `data_format` 为 `"channels_last"`，输入 5D 张量，尺寸为 `(batch, dim1, dim2, dim3, channels)`。
- 如果 `data_format` 为 `"channels_first"`，输入 5D 张量，尺寸为 `(batch, channels, dim1, dim2, dim3)`。

## 输出尺寸

- 如果 `data_format` 为 `"channels_last"`，输出 5D 张量，尺寸为 `(batch, upsampled_dim1, upsampled_dim2, upsampled_dim3, channels)`。
- 如果 `data_format` 为 `"channels_first"`，输出 5D 张量，尺寸为 `(batch, channels, upsampled_dim1, upsampled_dim2, upsampled_dim3)`。

ZeroPadding1D

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

1D 输入的零填充层（例如，时间序列）。

## 参数

- **padding**: 整数，或长度为 2 的整数元组，或字典。
  - 如果为整数：在填充维度（第一个轴）的开始和结束处添加多少个零。
  - 如果是长度为 2 的整数元组：在填充维度的开始和结尾处添加多少个零 (`(left_pad, right_pad)`)。

## 输入尺寸

3D 张量，尺寸为 `(batch, axis_to_pad, features)`。

## 输出尺寸

3D 张量，尺寸为 `(batch, padded_axis, features)`。

## ZeroPadding2D

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

2D 输入的零填充层（例如图像）。

该图层可以在图像张量的顶部、底部、左侧和右侧添加零表示的行和列。

## 参数

- **padding**: 整数，或 2 个整数的元组，或 2 个整数的 2 个元组。

- 如果为整数：将对宽度和高度运用相同的对称填充。

- 如果为 2 个整数的元组：

- 如果为整数：解释为高度和宽度的 2 个不同的对称裁剪值：

- (symmetric\_height\_pad, symmetric\_width\_pad)。

- 如果为 2 个整数的 2 个元组：解释为 ((top\_pad, bottom\_pad), (left\_pad, right\_pad))。

- **data\_format**: 字符串，`channels_last` (默认) 或 `channels_first` 之一，表示输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, height, width, channels)`，`channels_first` 对应输入尺寸为 `(batch, channels, height, width)`。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用“`channels_last`”。

## 输入尺寸

- 如果 `data_format` 为 `"channels_last"`，输入 4D 张量，尺寸为 `(batch, rows, cols, channels)`。
- 如果 `data_format` 为 `"channels_first"`，输入 4D 张量，尺寸为 `(batch, channels, rows, cols)`。

## 输出尺寸

- 如果 `data_format` 为 `"channels_last"`，输出 4D 张量，尺寸为 `(batch, padded_rows, padded_cols, channels)`。
- 如果 `data_format` 为 `"channels_first"`，输出 4D 张量，尺寸为 `(batch, channels, padded_rows, padded_cols)`。

## ZeroPadding3D

[\[source\]](#)

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

3D 数据的零填充层(空间或时空)。

## 参数

- **padding**: 整数, 或 3 个整数的元组, 或 2 个整数的 3 个元组。
  - 如果为整数: 将对深度、高度和宽度运用相同的对称填充。
  - 如果为 3 个整数的元组: 解释为深度、高度和宽度的三个不同的对称填充值:  
`(symmetric_dim1_pad, symmetric_dim2_pad, symmetric_dim3_pad)`。
  - 如果为 2 个整数的 3 个元组: 解释为 `((left_dim1_pad, right_dim1_pad), (left_dim2_pad, right_dim2_pad), (left_dim3_pad, right_dim3_pad))`
- **data\_format**: 字符串, `channels_last` (默认) 或 `channels_first` 之一, 表示输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)`, `channels_first` 对应输入尺寸为 `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它, 将使用 "channels\_last"。

## 输入尺寸

5D 张量, 尺寸为:

- 如果 `data_format` 为 "`channels_last`": `(batch, first_axis_to_pad, second_axis_to_pad, third_axis_to_pad, depth)`。
- 如果 `data_format` 为 "`channels_first`": `(batch, depth, first_axis_to_pad, second_axis_to_pad, third_axis_to_pad)`。

## 输出尺寸

5D 张量, 尺寸为:

- 如果 `data_format` 为 "`channels_last`": `(batch, first_padded_axis, second_padded_axis, third_axis_to_pad, depth)`。
- 如果 `data_format` 为 "`channels_first`": `(batch, depth, first_padded_axis, second_padded_axis, third_axis_to_pad)`。



# 池化层 Pooling Layers

MaxPooling1D

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

对于时序数据的最大池化。

## 参数

- **pool\_size**: 整数，最大池化的窗口大小。
- **strides**: 整数，或者是 `None`。作为缩小比例的因数。例如，2 会使得输入张量缩小一半。  
如果是 `None`，那么默认值是 `pool_size`。
- **padding**: `"valid"` 或者 `"same"` (区分大小写)。
- **data\_format**: 字符串，`channels_last` (默认)或 `channels_first` 之一。表示输入各维度的顺序。`channels_last` 对应输入尺寸为 `(batch, steps, features)`，  
`channels_first` 对应输入尺寸为 `(batch, features, steps)`。

## 输入尺寸

- 如果 `data_format='channels_last'`， 输入为 3D 张量，尺寸为：  
`(batch_size, steps, features)`
- 如果 `data_format='channels_first'`， 输入为 3D 张量，尺寸为：  
`(batch_size, features, steps)`

## 输出尺寸

- 如果 `data_format='channels_last'`， 输出为 3D 张量，尺寸为：  
`(batch_size, downsampled_steps, features)`
- 如果 `data_format='channels_first'`， 输出为 3D 张量，尺寸为：  
`(batch_size, features, downsampled_steps)`

MaxPooling2D

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

对于空间数据的最大池化。

## 参数

- **pool\_size**: 整数，或者 2 个整数表示的元组，沿（垂直，水平）方向缩小比例的因数。  
(2, 2) 会把输入张量的两个维度都缩小一半。如果只使用一个整数，那么两个维度都会使用同样的窗口长度。
- **strides**: 整数，2 个整数表示的元组，或者是 `None`。表示步长值。如果是 `None`，那么默认值是 `pool_size`。
- **padding**: `"valid"` 或者 `"same"` (区分大小写)。
- **data\_format**: 字符串，`channels_last` (默认)或 `channels_first` 之一。表示输入各维度的顺序。`channels_last` 代表尺寸是 `(batch, height, width, channels)` 的输入张量，而 `channels_first` 代表尺寸是 `(batch, channels, height, width)` 的输入张量。默认值根据 Keras 配置文件 `~/.keras/keras.json` 中的 `image_data_format` 值来设置。如果没有设置过，那么默认值就是“`channels_last`”。

## 输入尺寸

- 如果 `data_format='channels_last'` : 尺寸是 `(batch_size, rows, cols, channels)` 的 4D 张量
- 如果 `data_format='channels_first'` : 尺寸是 `(batch_size, channels, rows, cols)` 的 4D 张量

## 输出尺寸

- 如果 `data_format='channels_last'` : 尺寸是  
`(batch_size, pooled_rows, pooled_cols, channels)` 的 4D 张量
- 如果 `data_format='channels_first'` : 尺寸是 `(batch_size, channels, pooled_rows, pooled_cols)` 的 4D 张量

MaxPooling3D

```
keras.layers.MaxPooling3D(pool_size=(2, 2, 2), strides=None, padding='valid',  
data_format=None)
```

对于 3D (空间, 或时空间) 数据的最大池化。

## 参数

- **pool\_size**: 3 个整数表示的元组, 缩小 (dim1, dim2, dim3) 比例的因数。 (2, 2, 2) 会把 3D 输入张量的每个维度缩小一半。
- **strides**: 3 个整数表示的元组, 或者是 `None`。步长值。
- **padding**: `"valid"` 或者 `"same"` (区分大小写)。
- **data\_format**: 字符串, `channels_last` (默认)或 `channels_first` 之一。 表示输入各维度的顺序。`channels_last` 代表尺寸是 `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` 的输入张量, 而 `channels_first` 代表尺寸是 `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)` 的输入张量。 默认值根据 Keras 配置文件 `~/.keras/keras.json` 中的 `image_data_format` 值来设置。如果还没有设置过, 那么默认值就是 “`channels_last`”。

## 输入尺寸

- 如果 `data_format='channels_last'` : 尺寸是 `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)` 的 5D 张量
- 如果 `data_format='channels_first'` : 尺寸是 `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)` 的 5D 张量

## 输出尺寸

- 如果 `data_format='channels_last'` : 尺寸是 `(batch_size, pooled_dim1, pooled_dim2, pooled_dim3, channels)` 的 5D 张量
- 如果 `data_format='channels_first'` : 尺寸是 `(batch_size, channels, pooled_dim1, pooled_dim2, pooled_dim3)` 的 5D 张量

AveragePooling1D

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

对于时序数据的平均池化。

## 参数

- **pool\_size**: 整数，平均池化的窗口大小。
- **strides**: 整数，或者是 `None`。作为缩小比例的因数。例如，2 会使得输入张量缩小一半。  
如果是 `None`，那么默认值是 `pool_size`。
- **padding**: `"valid"` 或者 `"same"` (区分大小写)。
- **data\_format**: 字符串，`channels_last` (默认)或 `channels_first` 之一。表示输入各维度的顺序。`channels_last` 对应输入尺寸为 `(batch, steps, features)`，  
`channels_first` 对应输入尺寸为 `(batch, features, steps)`。

## 输入尺寸

- 如果 `data_format='channels_last'`， 输入为 3D 张量，尺寸为：  
`(batch_size, steps, features)`
- 如果 `data_format='channels_first'`， 输入为 3D 张量，尺寸为： `(batch_size, features, steps)`

## 输出尺寸

- 如果 `data_format='channels_last'`， 输出为 3D 张量，尺寸为： `(batch_size, downsampled_steps, features)`
- 如果 `data_format='channels_first'`， 输出为 3D 张量，尺寸为： `(batch_size, features, downsampled_steps)`

AveragePooling2D

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

对于空间数据的平均池化。

## 参数

- **pool\_size**: 整数，或者 2 个整数表示的元组，沿（垂直，水平）方向缩小比例的因数。  
(2, 2) 会把输入张量的两个维度都缩小一半。如果只使用一个整数，那么两个维度都会使用同样的窗口长度。
- **strides**: 整数，2 个整数表示的元组，或者是 `None`。表示步长值。如果是 `None`，那么默认值是 `pool_size`。
- **padding**: `"valid"` 或者 `"same"` (区分大小写)。
- **data\_format**: 字符串，`channels_last` (默认)或 `channels_first` 之一。表示输入各维度的顺序。`channels_last` 代表尺寸是 `(batch, height, width, channels)` 的输入张量，而 `channels_first` 代表尺寸是 `(batch, channels, height, width)` 的输入张量。默认值根据 Keras 配置文件 `~/.keras/keras.json` 中的 `image_data_format` 值来设置。如果没有设置过，那么默认值就是“`channels_last`”。

## 输入尺寸

- 如果 `data_format='channels_last'` : 尺寸是 `(batch_size, rows, cols, channels)` 的 4D 张量
- 如果 `data_format='channels_first'` : 尺寸是 `(batch_size, channels, rows, cols)` 的 4D 张量

## 输出尺寸

- 如果 `data_format='channels_last'` : 尺寸是  
`(batch_size, pooled_rows, pooled_cols, channels)` 的 4D 张量
- 如果 `data_format='channels_first'` : 尺寸是 `(batch_size, channels, pooled_rows, pooled_cols)` 的 4D 张量

AveragePooling3D

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

对于 3D (空间, 或者时空间) 数据的平均池化。

## 参数

- **pool\_size**: 3 个整数表示的元组, 缩小 (dim1, dim2, dim3) 比例的因数。 (2, 2, 2) 会把 3D 输入张量的每个维度缩小一半。
- **strides**: 3 个整数表示的元组, 或者 `None`。步长值。
- **padding**: `"valid"` 或者 `"same"` (区分大小写)。
- **data\_format**: 字符串, `channels_last` (默认)或 `channels_first` 之一。 表示输入各维度的顺序。`channels_last` 代表尺寸是 `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` 的输入张量, 而 `channels_first` 代表尺寸是 `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)` 的输入张量。 默认值根据 Keras 配置文件 `~/.keras/keras.json` 中的 `image_data_format` 值来设置。如果还没有设置过, 那么默认值就是 “`channels_last`”。

## 输入尺寸

- 如果 `data_format='channels_last'` : 尺寸是 `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)` 的 5D 张量
- 如果 `data_format='channels_first'` : 尺寸是 `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)` 的 5D 张量

## 输出尺寸

- 如果 `data_format='channels_last'` : 尺寸是 `(batch_size, pooled_dim1, pooled_dim2, pooled_dim3, channels)` 的 5D 张量
- 如果 `data_format='channels_first'` : 尺寸是 `(batch_size, channels, pooled_dim1, pooled_dim2, pooled_dim3)` 的 5D 张量

GlobalMaxPooling1D

```
keras.layers.GlobalMaxPooling1D(data_format='channels_last')
```

对于时序数据的全局最大池化。

## 参数

- **data\_format**: 字符串，`channels_last` (默认)或 `channels_first` 之一。表示输入各维度的顺序。`channels_last` 对应输入尺寸为 `(batch, steps, features)`，`channels_first` 对应输入尺寸为 `(batch, features, steps)`。

## 输入尺寸

尺寸是 `(batch_size, steps, features)` 的 3D 张量。

## 输出尺寸

尺寸是 `(batch_size, features)` 的 2D 张量。

GlobalAveragePooling1D

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

对于时序数据的全局平均池化。

### 输入尺寸

- 如果 `data_format='channels_last'`， 输入为 3D 张量，尺寸为：  
`(batch_size, steps, features)`
- 如果 `data_format='channels_first'`， 输入为 3D 张量，尺寸为： `(batch_size, features, steps)`

### 输出尺寸

尺寸是 `(batch_size, features)` 的 2D 张量。

GlobalMaxPooling2D

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

对于空域数据的全局最大池化。

## 参数

- **data\_format**: 字符串, `channels_last` (默认)或 `channels_first` 之一。表示输入各维度的顺序。`channels_last` 代表尺寸是 `(batch, height, width, channels)` 的输入张量, 而 `channels_first` 代表尺寸是 `(batch, channels, height, width)` 的输入张量。默认值根据 Keras 配置文件 `~/.keras/keras.json` 中的 `image_data_format` 值来设置。如果没有设置过, 那么默认值就是“`channels_last`”。

## 输入尺寸

- 如果 `data_format='channels_last'`: 尺寸是 `(batch_size, rows, cols, channels)` 的 4D 张量
- 如果 `data_format='channels_first'`: 尺寸是 `(batch_size, channels, rows, cols)` 的 4D 张量

## 输出尺寸

尺寸是 `(batch_size, channels)` 的 2D 张量

GlobalAveragePooling2D

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

对于空域数据的全局平均池化。

## 参数

- **data\_format**: 一个字符串，`channels_last`（默认值）或者`channels_first`。输入张量中的维度顺序。`channels_last`代表尺寸是`(batch, height, width, channels)`的输入张量，而`channels_first`代表尺寸是`(batch, channels, height, width)`的输入张量。默认值根据Keras配置文件`~/.keras/keras.json`中的`image_data_format`值来设置。如果还没有设置过，那么默认值就是“`channels_last`”。

## 输入尺寸

- 如果`data_format='channels_last'`: 尺寸是`(batch_size, rows, cols, channels)`的4D张量
- 如果`data_format='channels_first'`: 尺寸是`(batch_size, channels, rows, cols)`的4D张量

## 输出尺寸

尺寸是`(batch_size, channels)`的2D张量

## GlobalMaxPooling3D

```
keras.layers.GlobalMaxPooling3D(data_format=None)
```

对于 3D 数据的全局最大池化。

### 参数

- **data\_format**: 字符串，`channels_last` (默认)或 `channels_first` 之一。表示输入各维度的顺序。`channels_last` 代表尺寸是 `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` 的输入张量，而 `channels_first` 代表尺寸是 `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)` 的输入张量。默认值根据 Keras 配置文件 `~/.keras/keras.json` 中的 `image_data_format` 值来设置。如果还没有设置过，那么默认值就是“`channels_last`”。

## 输入尺寸

- 如果 `data_format='channels_last'` : 尺寸是 `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)` 的 5D 张量
- 如果 `data_format='channels_first'` : 尺寸是 `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)` 的 5D 张量

## 输出尺寸

尺寸是 `(batch_size, channels)` 的 2D 张量

## GlobalAveragePooling3D

[\[source\]](#)

```
keras.layers.GlobalAveragePooling3D(data_format=None)
```

对于 3D 数据的全局平均池化。

## 参数

- **data\_format**: 字符串, `channels_last` (默认)或 `channels_first` 之一。表示输入各维度的顺序。`channels_last` 代表尺寸是 `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` 的输入张量, 而 `channels_first` 代表尺寸是 `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)` 的输入张量。默认值根据 Keras 配置文件 `~/.keras/keras.json` 中的 `image_data_format` 值来设置。如果还没有设置过, 那么默认值就是 “`channels_last`”。

## 输入尺寸

- 如果 `data_format='channels_last'` : 尺寸是 `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)` 的 5D 张量
- 如果 `data_format='channels_first'` : 尺寸是 `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)` 的 5D 张量

## 输出尺寸

尺寸是 `(batch_size, channels)` 的 2D 张量



# 局部连接层 Locally-connected Layers

LocallyConnected1D

```
keras.layers.LocallyConnected1D(filters, kernel_size, strides=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)
```

1D 输入的局部连接层。

`LocallyConnected1D` 层与 `Conv1D` 层的工作方式相同，除了权值不共享外，也就是说，在输入的每个不同部分应用不同的一组过滤器。

## 示例

```
# 将长度为 3 的非共享权重 1D 卷积应用于
# 具有 10 个时间步长的序列，并使用 64个 输出滤波器
model = Sequential()
model.add(LocallyConnected1D(64, 3, input_shape=(10, 32)))
# 现在 model.output_shape == (None, 8, 64)
# 在上面再添加一个新的 conv1d
model.add(LocallyConnected1D(32, 3))
# 现在 model.output_shape == (None, 6, 32)
```

## 参数

- **filters**: 整数，输出空间的维度（即卷积中滤波器的输出数量）。
- **kernel\_size**: 一个整数，或者单个整数表示的元组或列表，指明 1D 卷积窗口的长度。
- **strides**: 一个整数，或者单个整数表示的元组或列表，指明卷积的步长。指定任何 `stride!=1` 与指定 `dilation_rate!=1` 两者不兼容。
- **padding**: 当前仅支持 `"valid"` (大小写敏感)。`"same"` 可能会在未来支持。
- **data\_format**: 字符串，`channels_first`, `channels_last` 之一。
- **activation**: 要使用的激活函数 (详见 [activations](#))。如果你不指定，则不使用激活函数 (即线性激活： $a(x) = x$ )。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。

## 输入尺寸

3D 张量，尺寸为：`(batch_size, steps, input_dim)`。

## 输出尺寸

3D 张量，尺寸为：`(batch_size, new_steps, filters)`，`steps` 值可能因填充或步长而改变。

## LocallyConnected2D

[\[source\]](#)

```
keras.layers.LocallyConnected2D(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)
```

2D 输入的局部连接层。

`LocallyConnected2D` 层与 `Conv2D` 层的工作方式相同，除了权值不共享外，也就是说，在输入的每个不同部分应用不同的一组过滤器。

## 示例

```
# 在 32x32 图像上应用 3x3 非共享权值和64个输出过滤器的卷积
# 数据格式 `data_format="channels_last"`:
model = Sequential()
model.add(LocallyConnected2D(64, (3, 3), input_shape=(32, 32, 3)))
# 现在 model.output_shape == (None, 30, 30, 64)
# 注意这一层的参数数量为 (30*30)*(3*3*3*64) + (30*30)*64

# 在上面再加一个 3x3 非共享权值和 32 个输出滤波器的卷积:
model.add(LocallyConnected2D(32, (3, 3)))
# 现在 model.output_shape == (None, 28, 28, 32)
```

## 参数

- **filters**: 整数，输出空间的维度（即卷积中滤波器的输出数量）。
- **kernel\_size**: 一个整数，或者 2 个整数表示的元组或列表，指明 2D 卷积窗口的宽度和高度。可以是一个整数，为所有空间维度指定相同的值。
- **strides**: 一个整数，或者 2 个整数表示的元组或列表，指明卷积沿宽度和高度方向的步长。可以是一个整数，为所有空间维度指定相同的值。
- **padding**: 当前仅支持 "valid" (大小写敏感)。"same" 可能会在未来支持。
- **data\_format**: 字符串，`channels_last` (默认) 或 `channels_first` 之一。输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, height, width, channels)`，`channels_first` 对应输入尺寸为 `(batch, channels, height, width)`。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用 "channels\_last"。
- **activation**: 要使用的激活函数 (详见 [activations](#))。如果你不指定，则不使用激活函数 (即线性激活:  $a(x) = x$ )。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。

## 输入尺寸

4D 张量，尺寸为：`(samples, channels, rows, cols)`，如果  
`data_format='channels_first'`；或者 4D 张量，尺寸为：`(samples, rows, cols, channels)`，如果 `data_format='channels_last'`。

## 输出尺寸

4D 张量，尺寸为：`(samples, filters, new_rows, new_cols)`，如果  
`data_format='channels_first'`；或者 4D 张量，尺寸为：`(samples, new_rows, new_cols, filters)`，如果 `data_format='channels_last'`。`rows` 和 `cols` 的值可能因填充而改变。



# 循环层 Recurrent Layers

RNN

```
keras.layers.RNN(cell, return_sequences=False, return_state=False,  
go_backwards=False, stateful=False, unroll=False)
```

循环神经网络层基类。

## 参数

- **cell**: 一个 RNN 单元实例。RNN 单元是一个具有以下几项的类：
  - 一个 `call(input_at_t, states_at_t)` 方法，它返回 `(output_at_t, states_at_t_plus_1)`。单元的调用方法也可以采引入可选参数 `constants`，详见下面的小节「关于给 RNN 传递外部常量的说明」。
  - 一个 `state_size` 属性。这可以是单个整数（单个状态），在这种情况下，它是循环层状态的大小（应该与单元输出的大小相同）。这也可以是整数表示的列表/元组（每个状态一个大小）。
  - 一个 `output_size` 属性。这可以是单个整数或者是一个 `TensorShape`，它表示输出的尺寸。出于向后兼容的原因，如果此属性对于当前单元不可用，则该值将由 `state_size` 的第一个元素推断。
- `cell` 也可能是 RNN 单元实例的列表，在这种情况下，RNN 的单元将堆叠在另一个单元上，实现高效的堆叠 RNN。
- **return\_sequences**: 布尔值。是返回输出序列中的最后一个输出，还是全部序列。
- **return\_state**: 布尔值。除了输出之外是否返回最后一个状态。
- **go\_backwards**: 布尔值 (默认 False)。如果为 True，则向后处理输入序列并返回相反的序列。
- **stateful**: 布尔值 (默认 False)。如果为 True，则批次中索引  $i$  处的每个样品的最后状态将用作下一批次中索引  $i$  样品的初始状态。
- **unroll**: 布尔值 (默认 False)。如果为 True，则网络将展开，否则将使用符号循环。展开可以加速 RNN，但它往往会使占用更多的内存。展开只适用于短序列。
- **input\_dim**: 输入的维度 (整数)。将此层用作模型中的第一层时，此参数 (或者，关键字参数 `input_shape`) 是必需的。
- **input\_length**: 输入序列的长度，在恒定时指定。如果你要在上游连接 `Flatten` 和 `Dense` 层，则需要此参数 (如果没有它，无法计算全连接输出的尺寸)。请注意，如果循环神经网络层不是模型中的第一层，则需要在第一层的层级指定输入长度 (例如，通过 `input_shape` 参数)。

## 输入尺寸

3D 张量，尺寸为 `(batch_size, timesteps, input_dim)`。

## 输出尺寸

- 如果 `return_state`：返回张量列表。第一个张量为输出。剩余的张量为最后的状态，每个张量的尺寸为 `(batch_size, units)`。例如，对于 RNN/GRU，状态张量数目为 1，对 LSTM 为 2。

- 如果 `return_sequences`：返回 3D 张量，尺寸为 `(batch_size, timesteps, units)`。
- 否则，返回尺寸为 `(batch_size, units)` 的 2D 张量。

## Masking

该层支持以可变数量的时间步对输入数据进行 masking。要将 masking 引入你的数据，请使用 [Embedding](#) 层，并将 `mask_zero` 参数设置为 `True`。

## 关于在 RNN 中使用「状态（statefulness）」的说明

你可以将 RNN 层设置为 `stateful`（有状态的），这意味着针对一个批次的样本计算的状态将被重新用作下一批样本的初始状态。这假定在不同连续批次的样品之间有一对一的映射。

为了使状态有效：

- 在层构造器中指定 `stateful=True`。
- 为你的模型指定一个固定的批次大小，如果是顺序模型，为你的模型的第一层传递一个 `batch_input_shape=(...)` 参数。
- 为你的模型指定一个固定的批次大小，如果是顺序模型，为你的模型的第一层传递一个 `batch_input_shape=(...)`。如果是带有 1 个或多个 Input 层的函数式模型，为你的模型的所有第一层传递一个 `batch_shape=(...)`。这是你的输入的预期尺寸，包括批量维度。它应该是整数的元组，例如 `(32, 10, 100)`。
- 在调用 `fit()` 是指定 `shuffle=False`。

要重置模型的状态，请在特定图层或整个模型上调用 `.reset_states()`。

## 关于指定 RNN 初始状态的说明

您可以通过使用关键字参数 `initial_state` 调用它们来符号化地指定 RNN 层的初始状态。`initial_state` 的值应该是表示 RNN 层初始状态的张量或张量列表。

您可以通过调用带有关键字参数 `states` 的 `reset_states` 方法来数字化地指定 RNN 层的初始状态。`states` 的值应该是一个代表 RNN 层初始状态的 Numpy 数组或者 Numpy 数组列表。

## 关于给 RNN 传递外部常量的说明

你可以使用 `RNN.__call__`（以及 `RNN.call`）的 `constants` 关键字参数将「外部」常量传递给单元。这要求 `cell.call` 方法接受相同的关键字参数 `constants`。这些常数可用于调节附加静态输入（不随时间变化）上的单元转换，也可用于注意力机制。

## 示例

```
# 首先，让我们定义一个 RNN 单元，作为网络层子类。
```

```
class MinimalRNNCell(keras.layers.Layer):
```

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

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

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

# 让我们在 RNN 层使用这个单元:

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

# 以下是如何使用单元格构建堆叠的 RNN的方法:

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



SimpleRNN

```
keras.layers.SimpleRNN(units, activation='tanh', use_bias=True,  
kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',  
bias_initializer='zeros', kernel_regularizer=None, recurrent_regularizer=None,  
bias_regularizer=None, activity_regularizer=None, kernel_constraint=None,  
recurrent_constraint=None, bias_constraint=None, dropout=0.0,  
recurrent_dropout=0.0, return_sequences=False, return_state=False,  
go_backwards=False, stateful=False, unroll=False)
```

全连接的 RNN，其输出将被反馈到输入。

## 参数

- **units**: 正整数，输出空间的维度。
- **activation**: 要使用的激活函数 (详见 [activations](#))。默认：双曲正切 (`tanh`)。如果传入 `None`，则不使用激活函数 (即 线性激活： $a(x) = x$ )。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器，用于输入的线性转换 (详见 [initializers](#))。
- **recurrent\_initializer**: `recurrent_kernel` 权值矩阵的初始化器，用于循环层状态的线性转换 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **recurrent\_regularizer**: 运用到 `recurrent_kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **recurrent\_constraint**: 运用到 `recurrent_kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。
- **dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于输入的线性转换。
- **recurrent\_dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于循环层状态的线性转换。
- **return\_sequences**: 布尔值。是返回输出序列中的最后一个输出，还是全部序列。
- **return\_state**: 布尔值。除了输出之外是否返回最后一个状态。
- **go\_backwards**: 布尔值 (默认 `False`)。如果为 `True`，则向后处理输入序列并返回相反的序列。
- **stateful**: 布尔值 (默认 `False`)。如果为 `True`，则批次中索引  $i$  处的每个样品的最后状态将用作下一批次中索引  $i$  样品的初始状态。
- **unroll**: 布尔值 (默认 `False`)。如果为 `True`，则网络将展开，否则将使用符号循环。展开可以加速 RNN，但它往往需要占用更多的内存。展开只适用于短序列。

GRU

```
keras.layers.GRU(units, activation='tanh', recurrent_activation='sigmoid',
use_bias=True, kernel_initializer='glorot_uniform',
recurrent_initializer='orthogonal', bias_initializer='zeros',
kernel_regularizer=None, recurrent_regularizer=None, bias_regularizer=None,
activity_regularizer=None, kernel_constraint=None, recurrent_constraint=None,
bias_constraint=None, dropout=0.0, recurrent_dropout=0.0, implementation=2,
return_sequences=False, return_state=False, go_backwards=False,
stateful=False, unroll=False, reset_after=False)
```

门限循环单元网络 (Gated Recurrent Unit) - Cho et al. 2014.

有两种变体。默认的是基于 1406.1078v3 的实现，同时在矩阵乘法之前将复位门应用于隐藏状态。另一种则是基于 1406.1078v1 的实现，它包括顺序倒置的操作。

第二种变体与 CuDNNGRU(GPU-only) 兼容并且允许在 CPU 上进行推理。因此它对于 `kernel` 和 `recurrent_kernel` 有可分离偏置。使用 `'reset_after'=True` 和 `recurrent_activation='sigmoid'`。

## 参数

- **units**: 正整数，输出空间的维度。
- **activation**: 要使用的激活函数 (详见 [activations](#))。默认：双曲正切 (`tanh`)。如果传入 `None`，则不使用激活函数 (即 线性激活：`a(x) = x`)。
- **recurrent\_activation**: 用于循环时间步的激活函数 (详见 [activations](#))。默认：分段线性近似 `sigmoid` (`hard_sigmoid`)。如果传入 `None`，则不使用激活函数 (即 线性激活：`a(x) = x`)。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器，用于输入的线性转换 (详见 [initializers](#))。
- **recurrent\_initializer**: `recurrent_kernel` 权值矩阵的初始化器，用于循环层状态的线性转换 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **recurrent\_regularizer**: 运用到 `recurrent_kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **recurrent\_constraint**: 运用到 `recurrent_kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。
- **dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于输入的线性转换。
- **recurrent\_dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于循环层状态的线性转换。
- **implementation**: 实现模式，1 或 2。模式 1 将把它的操作结构化为更多的小的点积和加法操作，而模式 2 将把它们分批到更少，更大的操作中。这些模式在不同的硬件和不同的应用中具有不同的性能配置文件。
- **return\_sequences**: 布尔值。是返回输出序列中的最后一个输出，还是全部序列。

- **return\_state**: 布尔值。除了输出之外是否返回最后一个状态。
- **go\_backwards**: 布尔值 (默认 False)。如果为 True，则向后处理输入序列并返回相反的序列。
- **stateful**: 布尔值 (默认 False)。如果为 True，则批次中索引  $i$  处的每个样品的最后状态 将用作下一批次中索引  $i$  样品的初始状态。
- **unroll**: 布尔值 (默认 False)。如果为 True，则网络将展开，否则将使用符号循环。展开可以加速 RNN，但它往往占用更多的内存。展开只适用于短序列。
- **reset\_after**:
- GRU 公约 (是否在矩阵乘法之前或者之后使用重置门)。False = 「之前」(默认)，True = 「之后」(CuDNN 兼容)。

## 参考文献

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

LSTM

```
keras.layers.LSTM(units, activation='tanh', recurrent_activation='sigmoid',
use_bias=True, kernel_initializer='glorot_uniform',
recurrent_initializer='orthogonal', bias_initializer='zeros',
unit_forget_bias=True, kernel_regularizer=None, recurrent_regularizer=None,
bias_regularizer=None, activity_regularizer=None, kernel_constraint=None,
recurrent_constraint=None, bias_constraint=None, dropout=0.0,
recurrent_dropout=0.0, implementation=2, return_sequences=False,
return_state=False, go_backwards=False, stateful=False, unroll=False)
```

长短期记忆网络层（Long Short-Term Memory） - Hochreiter 1997.

## 参数

- **units**: 正整数，输出空间的维度。
- **activation**: 要使用的激活函数 (详见 [activations](#))。如果传入 `None`，则不使用激活函数 (即线性激活:  $a(x) = x$ )。
- **recurrent\_activation**: 用于循环时间步的激活函数 (详见 [activations](#))。默认: 分段线性近似 sigmoid (`hard_sigmoid`)。如果传入 `None`，则不使用激活函数 (即 线性激活:  $a(x) = x$ )。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器，用于输入的线性转换 (详见 [initializers](#))。
- **recurrent\_initializer**: `recurrent_kernel` 权值矩阵的初始化器，用于循环层状态的线性转换 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **unit\_forget\_bias**: 布尔值。如果为 `True`，初始化时，将忘记门的偏置加 1。将其设置为 `True` 同时还会强制 `bias_initializer="zeros"`。这个建议来自 [Jozefowicz et al. \(2015\)](#)。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **recurrent\_regularizer**: 运用到 `recurrent_kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **recurrent\_constraint**: 运用到 `recurrent_kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。
- **dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于输入的线性转换。
- **recurrent\_dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于循环层状态的线性转换。
- **implementation**: 实现模式，1 或 2。模式 1 将把它的操作结构化为更多的小的点积和加法操作，而模式 2 将把它们分批到更少，更大的操作中。这些模式在不同的硬件和不同的应用中具有不同的性能配置文件。
- **return\_sequences**: 布尔值。是返回输出序列中的最后一个输出，还是全部序列。
- **return\_state**: 布尔值。除了输出之外是否返回最后一个状态。状态列表的返回元素分别是隐藏状态和单元状态。

- **go\_backwards**: 布尔值 (默认 False)。如果为 True，则向后处理输入序列并返回相反的序列。
- **stateful**: 布尔值 (默认 False)。如果为 True，则批次中索引  $i$  处的每个样品的最后状态 将用作下一批次中索引  $i$  样品的初始状态。
- **unroll**: 布尔值 (默认 False)。如果为 True，则网络将展开，否则将使用符号循环。展开可以加速 RNN，但它往往会占用更多的内存。展开只适用于短序列。

## 参考文献

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

ConvLSTM2D

```
keras.layers.ConvLSTM2D(filters, kernel_size, strides=(1, 1), padding='valid',
data_format=None, dilation_rate=(1, 1), activation='tanh',
recurrent_activation='hard_sigmoid', use_bias=True,
kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',
bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None,
recurrent_regularizer=None, bias_regularizer=None, activity_regularizer=None,
kernel_constraint=None, recurrent_constraint=None, bias_constraint=None,
return_sequences=False, go_backwards=False, stateful=False, dropout=0.0,
recurrent_dropout=0.0)
```

卷积 LSTM。

它类似于 LSTM 层，但输入变换和循环变换都是卷积的。

## 参数

- **filters**: 整数，输出空间的维度（即卷积中滤波器的输出数量）。
- **kernel\_size**: 一个整数，或者 n 个整数表示的元组或列表，指明卷积窗口的维度。
- **strides**: 一个整数，或者 n 个整数表示的元组或列表，指明卷积的步长。指定任何 stride 值 != 1 与指定 `dilation_rate` 值 != 1 两者不兼容。
- **padding**: `"valid"` 或 `"same"` 之一 (大小写敏感)。
- **data\_format**: 字符串，`channels_last` (默认) 或 `channels_first` 之一。输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, time, ..., channels)`，`channels_first` 对应输入尺寸为 `(batch, time, channels, ...)`。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用 `"channels_last"`。
- **dilation\_rate**: 一个整数，或 n 个整数的元组/列表，指定用于膨胀卷积的膨胀率。目前，指定任何 `dilation_rate` 值 != 1 与指定 `stride` 值 != 1 两者不兼容。
- **activation**: 要使用的激活函数 (详见 [activations](#))。如果传入 `None`，则不使用激活函数 (即线性激活:  $a(x) = x$ )。
- **recurrent\_activation**: 用于循环时间步的激活函数 (详见 [activations](#))。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器，用于输入的线性转换 (详见 [initializers](#))。
- **recurrent\_initializer**: `recurrent_kernel` 权值矩阵的初始化器，用于循环层状态的线性转换 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **unit\_forget\_bias**: 布尔值。如果为 `True`，初始化时，将忘记门的偏置加 1。将其设置为 `True` 同时还会强制 `bias_initializer="zeros"`。这个建议来自 [Jozefowicz et al. \(2015\)](#)。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **recurrent\_regularizer**: 运用到 `recurrent_kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **recurrent\_constraint**: 运用到 `recurrent_kernel` 权值矩阵的约束函数 (详见 [constraints](#))。

- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。
- **return\_sequences**: 布尔值。是返回输出序列中的最后一个输出，还是全部序列。
- **go\_backwards**: 布尔值 (默认 False)。如果为 True，则向后处理输入序列并返回相反的序列。
- **stateful**: 布尔值 (默认 False)。如果为 True，则批次中索引  $i$  处的每个样品的最后状态 将用作下一批次中索引  $i$  样品的初始状态。
- **dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于输入的线性转换。
- **recurrent\_dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于循环层状态的线性转换。

## 输入尺寸

- 如果 `data_format='channels_first'`， 输入 5D 张量，尺寸为：  
`(samples, time, channels, rows, cols)`。
- 如果 `data_format='channels_last'`， 输入 5D 张量，尺寸为： `(samples, time, rows, cols, channels)`。

## 输出尺寸

- 如果 `return_sequences`，
  - 如果 `data_format='channels_first'`，返回 5D 张量，尺寸为： `(samples, time, filters, output_row, output_col)`。
  - 如果 `data_format='channels_last'`，返回 5D 张量，尺寸为： `(samples, time, output_row, output_col, filters)`。
- 否则，
  - 如果 `data_format = 'channels_first'`，返回 4D 张量，尺寸为： `(samples, filters, output_row, output_col)`。
  - 如果 `data_format='channels_last'`，返回 4D 张量，尺寸为： `(samples, output_row, output_col, filters)`。

`o_row` 和 `o_col` 取决于 `filter` 和 `padding` 的尺寸。

## 异常

- **ValueError**: 无效的构造参数。

## 参考文献

- [Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting](#)。当前的实现不包括单元输出的反馈回路。

ConvLSTM2DCell

```
keras.layers.ConvLSTM2DCell(filters, kernel_size, strides=(1, 1),
padding='valid', data_format=None, dilation_rate=(1, 1), activation='tanh',
recurrent_activation='hard_sigmoid', use_bias=True,
kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',
bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None,
recurrent_regularizer=None, bias_regularizer=None, kernel_constraint=None,
recurrent_constraint=None, bias_constraint=None, dropout=0.0,
recurrent_dropout=0.0)
```

ConvLSTM2D 层的单元类。

## 参数

- **filters**: 整数，输出空间的维度（即卷积中滤波器的输出数量）。
- **kernel\_size**: 一个整数，或者 n 个整数表示的元组或列表，指明卷积窗口的维度。
- **strides**: 一个整数，或者 n 个整数表示的元组或列表，指明卷积的步长。指定任何 stride 值 != 1 与指定 `dilation_rate` 值 != 1 两者不兼容。
- **padding**: `"valid"` 或 `"same"` 之一 (大小写敏感)。
- **data\_format**: 字符串，`channels_last` (默认) 或 `channels_first` 之一。输入中维度的顺序。`channels_last` 对应输入尺寸为 `(batch, time, ..., channels)`，`channels_first` 对应输入尺寸为 `(batch, time, channels, ...)`。它默认为从 Keras 配置文件 `~/.keras/keras.json` 中找到的 `image_data_format` 值。如果你从未设置它，将使用 `"channels_last"`。
- **dilation\_rate**: 一个整数，或 n 个整数的元组/列表，指定用于膨胀卷积的膨胀率。目前，指定任何 `dilation_rate` 值 != 1 与指定 `stride` 值 != 1 两者不兼容。
- **activation**: 要使用的激活函数 (详见 [activations](#))。如果传入 `None`，则不使用激活函数 (即线性激活： $a(x) = x$ )。
- **recurrent\_activation**: 用于循环时间步的激活函数 (详见 [activations](#))。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器，用于输入的线性转换 (详见 [initializers](#))。
- **recurrent\_initializer**: `recurrent_kernel` 权值矩阵的初始化器，用于循环层状态的线性转换 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **unit\_forget\_bias**: 布尔值。如果为 `True`，初始化时，将忘记门的偏置加 1。将其设置为 `True` 同时还会强制 `bias_initializer="zeros"`。这个建议来自 [Jozefowicz et al. \(2015\)](#)。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **recurrent\_regularizer**: 运用到 `recurrent_kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **recurrent\_constraint**: 运用到 `recurrent_kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。
- **dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于输入的线性转换。

- **recurrent\_dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于循环层状态的线性转换。

SimpleRNNCell

```
keras.layers.SimpleRNNCell(units, activation='tanh', use_bias=True,  
kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',  
bias_initializer='zeros', kernel_regularizer=None, recurrent_regularizer=None,  
bias_regularizer=None, kernel_constraint=None, recurrent_constraint=None,  
bias_constraint=None, dropout=0.0, recurrent_dropout=0.0)
```

SimpleRNN 的单元类。

## 参数

- **units**: 正整数，输出空间的维度。
- **activation**: 要使用的激活函数 (详见 [activations](#))。默认：双曲正切 (`tanh`)。如果传入 `None`，则不使用激活函数 (即 线性激活： $a(x) = x$ )。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器，用于输入的线性转换 (详见 [initializers](#))。
- **recurrent\_initializer**: `recurrent_kernel` 权值矩阵的初始化器，用于循环层状态的线性转换 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **recurrent\_regularizer**: 运用到 `recurrent_kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **recurrent\_constraint**: 运用到 `recurrent_kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。
- **dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于输入的线性转换。
- **recurrent\_dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于循环层状态的线性转换。

GRUCell

```
keras.layers.GRUCell(units, activation='tanh', recurrent_activation='sigmoid',
use_bias=True, kernel_initializer='glorot_uniform',
recurrent_initializer='orthogonal', bias_initializer='zeros',
kernel_regularizer=None, recurrent_regularizer=None, bias_regularizer=None,
kernel_constraint=None, recurrent_constraint=None, bias_constraint=None,
dropout=0.0, recurrent_dropout=0.0, implementation=2, reset_after=False)
```

GRU 层的单元类。

## 参数

- **units**: 正整数，输出空间的维度。
- **activation**: 要使用的激活函数 (详见 [activations](#))。默认：双曲正切 (`tanh`)。如果传入 `None`，则不使用激活函数 (即 线性激活： $a(x) = x$ )。
- **recurrent\_activation**: 用于循环时间步的激活函数 (详见 [activations](#))。默认：分段线性近似 `sigmoid` (`hard_sigmoid`)。如果传入 `None`，则不使用激活函数 (即 线性激活： $a(x) = x$ )。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器，用于输入的线性转换 (详见 [initializers](#))。
- **recurrent\_initializer**: `recurrent_kernel` 权值矩阵的初始化器，用于循环层状态的线性转换 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **recurrent\_regularizer**: 运用到 `recurrent_kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **recurrent\_constraint**: 运用到 `recurrent_kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。
- **dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于输入的线性转换。
- **recurrent\_dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于循环层状态的线性转换。
- **implementation**: 实现模式，1 或 2。模式 1 将把它的操作结构化为更多的小的点积和加法操作，而模式 2 将把它们分批到更少，更大的操作中。这些模式在不同的硬件和不同的应用中具有不同的性能配置文件。
- **reset\_after**:
  - GRU 公约 (是否在矩阵乘法之前或者之后使用重置门)。`False` = “before” (默认)，`True` = “after” (CuDNN 兼容)。
  - **reset\_after**: GRU convention (whether to apply reset gate after or before matrix multiplication). `False` = “before” (default), `True` = “after” (CuDNN compatible).

LSTMCell

```
keras.layers.LSTMCell(units, activation='tanh',
recurrent_activation='sigmoid', use_bias=True,
kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',
bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None,
recurrent_regularizer=None, bias_regularizer=None, kernel_constraint=None,
recurrent_constraint=None, bias_constraint=None, dropout=0.0,
recurrent_dropout=0.0, implementation=2)
```

LSTM 层的单元类。

## 参数

- **units**: 正整数，输出空间的维度。
- **activation**: 要使用的激活函数 (详见 [activations](#))。默认：双曲正切 (`tanh`)。如果传入 `None`，则不使用激活函数 (即 线性激活： $a(x) = x$ )。
- **recurrent\_activation**: 用于循环时间步的激活函数 (详见 [activations](#))。默认：分段线性近似 `sigmoid` (`hard_sigmoid`)。
- **use\_bias**: 布尔值，该层是否使用偏置向量。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器，用于输入的线性转换 (详见 [initializers](#))。
- **recurrent\_initializer**: `recurrent_kernel` 权值矩阵的初始化器，用于循环层状态的线性转换 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **unit\_forget\_bias**: 布尔值。如果为 `True`，初始化时，将忘记门的偏置加 1。将其设置为 `True` 同时还会强制 `bias_initializer="zeros"`。这个建议来自 [Jozefowicz et al. \(2015\)](#)。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **recurrent\_regularizer**: 运用到 `recurrent_kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **recurrent\_constraint**: 运用到 `recurrent_kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。
- **dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于输入的线性转换。
- **recurrent\_dropout**: 在 0 和 1 之间的浮点数。单元的丢弃比例，用于循环层状态的线性转换。
- **implementation**: 实现模式，1 或 2。模式 1 将把它的操作结构化为更多的小的点积和加法操作，而模式 2 将把它们分批到更少，更大的操作中。这些模式在不同的硬件和不同的应用中具有不同的性能配置文件。

## CuDNNGRU

```
keras.layers.CuDNNGRU(units, kernel_initializer='glorot_uniform',
recurrent_initializer='orthogonal', bias_initializer='zeros',
kernel_regularizer=None, recurrent_regularizer=None, bias_regularizer=None,
activity_regularizer=None, kernel_constraint=None, recurrent_constraint=None,
bias_constraint=None, return_sequences=False, return_state=False,
stateful=False)
```

由 CuDNN 支持的快速 GRU 实现。

只能以 TensorFlow 后端运行在 GPU 上。

## 参数

- **units**: 正整数，输出空间的维度。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器，用于输入的线性转换 (详见 [initializers](#))。
- **recurrent\_initializer**: `recurrent_kernel` 权值矩阵的初始化器，用于循环层状态的线性转换 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **recurrent\_regularizer**: 运用到 `recurrent_kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see [regularizer](#)).
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **recurrent\_constraint**: 运用到 `recurrent_kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。
- **return\_sequences**: 布尔值。是返回输出序列中的最后一个输出，还是全部序列。
- **return\_state**: 布尔值。除了输出之外是否返回最后一个状态。
- **stateful**: 布尔值 (默认 False)。如果为 True，则批次中索引 i 处的每个样品的最后状态 将用作下一批次中索引 i 样品的初始状态。

## CuDNNLSTM

[\[source\]](#)

```
keras.layers.CuDNNLSTM(units, kernel_initializer='glorot_uniform',
                        recurrent_initializer='orthogonal', bias_initializer='zeros',
                        unit_forget_bias=True, kernel_regularizer=None, recurrent_regularizer=None,
                        bias_regularizer=None, activity_regularizer=None, kernel_constraint=None,
                        recurrent_constraint=None, bias_constraint=None, return_sequences=False,
                        return_state=False, stateful=False)
```

由 CuDNN 支持的快速 LSTM 实现。

只能以 TensorFlow 后端运行在 GPU 上。

## 参数

- **units**: 正整数，输出空间的维度。
- **kernel\_initializer**: `kernel` 权值矩阵的初始化器，用于输入的线性转换 (详见 [initializers](#))。
- **recurrent\_initializer**: `recurrent_kernel` 权值矩阵的初始化器，用于循环层状态的线性转换 (详见 [initializers](#))。
- **bias\_initializer**: 偏置向量的初始化器 (详见 [initializers](#))。
- **unit\_forget\_bias**: 布尔值。如果为 True，初始化时，将忘记门的偏置加 1。将其设置为 True 同时还会强制 `bias_initializer="zeros"`。这个建议来自 [Jozefowicz et al. \(2015\)](#)。
- **kernel\_regularizer**: 运用到 `kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **recurrent\_regularizer**: 运用到 `recurrent_kernel` 权值矩阵的正则化函数 (详见 [regularizer](#))。
- **bias\_regularizer**: 运用到偏置向量的正则化函数 (详见 [regularizer](#))。
- **activity\_regularizer**: 运用到层输出 (它的激活值) 的正则化函数 (详见 [regularizer](#))。
- **kernel\_constraint**: 运用到 `kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **recurrent\_constraint**: 运用到 `recurrent_kernel` 权值矩阵的约束函数 (详见 [constraints](#))。
- **bias\_constraint**: 运用到偏置向量的约束函数 (详见 [constraints](#))。
- **return\_sequences**: 布尔值。是返回输出序列中的最后一个输出，还是全部序列。
- **return\_state**: 布尔值。除了输出之外是否返回最后一个状态。
- **stateful**: 布尔值 (默认 False)。如果为 True，则批次中索引  $i$  处的每个样品的最后状态 将用作下一批次中索引  $i$  样品的初始状态。



# 嵌入层 Embedding Layers

## Embedding

```
keras.layers.Embedding(input_dim, output_dim,  
embeddings_initializer='uniform', embeddings_regularizer=None,  
activity_regularizer=None, embeddings_constraint=None, mask_zero=False,  
input_length=None)
```

将正整数（索引值）转换为固定尺寸的稠密向量。例如： [[4], [20]] -> [[0.25, 0.1], [0.6, -0.2]]

该层只能用作模型中的第一层。

### 示例

```
model = Sequential()  
model.add(Embedding(1000, 64, input_length=10))  
# 模型将输入一个大小为 (batch, input_length) 的整数矩阵。  
# 输入中最大的整数（即词索引）不应该大于 999（词汇表大小）  
# 现在 model.output_shape == (None, 10, 64)，其中 None 是 batch 的维度。  
  
input_array = np.random.randint(1000, size=(32, 10))  
  
model.compile('rmsprop', 'mse')  
output_array = model.predict(input_array)  
assert output_array.shape == (32, 10, 64)
```

### 参数

- **input\_dim**: int > 0。词汇表大小，即，最大整数 index + 1。
- **output\_dim**: int >= 0。词向量的维度。
- **embeddings\_initializer**: embeddings 矩阵的初始化方法 (详见 [initializers](#))。
- **embeddings\_regularizer**: embeddings matrix 的正则化方法 (详见 [regularizer](#))。
- **activity\_regularizer**: 应用到层输出的正则化函数 (它的“activation”)。 (详见 [regularizer](#))。
- **embeddings\_constraint**: embeddings matrix 的约束函数 (详见 [constraints](#))。
- **mask\_zero**: 是否把 0 看作为一个应该被遮蔽的特殊的“padding”值。这对于可变长的[循环神经网络层](#)十分有用。如果设定为 `True`，那么接下来的所有层都必须支持 masking，否则就会抛出异常。如果 `mask_zero` 为 `True`，作为结果，索引 0 就不能被用于词汇表中 (`input_dim` 应该与 `vocabulary + 1` 大小相同)。
- **input\_length**: 输入序列的长度，当它是固定的时。如果你需要连接 `Flatten` 和 `Dense` 层，则这个参数是必须的（没有它，`dense` 层的输出尺寸就无法计算）。

## 输入尺寸

尺寸为 `(batch_size, sequence_length)` 的 2D 张量。

## 输出尺寸

尺寸为 `(batch_size, sequence_length, output_dim)` 的 3D 张量。

## 参考文献

- [A Theoretically Grounded Application of Dropout in Recurrent Neural Networks](#)



## 融合层 Merge Layers

Add

```
keras.layers.Add()
```

计算输入张量列表的和。

它接受一个张量的列表，所有的张量必须有相同的输入尺寸，然后返回一个张量（和输入张量尺寸相同）。

### 示例

```
import keras

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

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



**Subtract**

```
keras.layers.Subtract()
```

计算两个输入张量的差。

它接受一个长度为 2 的张量列表， 两个张量必须有相同的尺寸， 然后返回一个值为 (inputs[0] - inputs[1]) 的张量， 输出张量和输入张量尺寸相同。

### 示例

```
import keras

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

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



Multiply

```
keras.layers.Multiply()
```

计算输入张量列表的（逐元素间的）乘积。

它接受一个张量的列表，所有的张量必须有相同的输入尺寸，然后返回一个张量（和输入张量尺寸相同）。

Average

```
keras.layers.Average()
```

计算输入张量列表的平均值。

它接受一个张量的列表，所有的张量必须有相同的输入尺寸，然后返回一个张量（和输入张量尺寸相同）。

## Maximum

[\[source\]](#)

```
keras.layers.Maximum()
```

计算输入张量列表的（逐元素间的）最大值。

它接受一个张量的列表，所有的张量必须有相同的输入尺寸，然后返回一个张量（和输入张量尺寸相同）。

[\[source\]](#)

## Minimum

```
keras.layers.Minimum()
```

计算输入张量列表的（逐元素间的）最小值。

它接受一个张量的列表，所有的张量必须有相同的输入尺寸，然后返回一个张量（和输入张量尺寸相同）。

## Concatenate

[\[source\]](#)

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

连接一个输入张量的列表。

它接受一个张量的列表，除了连接轴之外，其他的尺寸都必须相同，然后返回一个由所有输入张量连接起来的输出张量。

## 参数

- **axis**: 连接的轴。
- **\*\*kwargs**: 层关键字参数。

## Dot

[\[source\]](#)

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

计算两个张量之间样本的点积。

例如，如果作用于输入尺寸为 `(batch_size, n)` 的两个张量 `a` 和 `b`，那么输出结果就会是尺寸为 `(batch_size, 1)` 的一个张量。在这个张量中，每一个条目 `i` 是 `a[i]` 和 `b[i]` 之间的点积。

## 参数

- **axes**: 整数或者整数元组，一个或者几个进行点积的轴。
- **normalize**: 是否在点积之前对即将进行点积的轴进行 L2 标准化。如果设置成 `True`，那么输出两个样本之间的余弦相似值。
- **\*\*kwargs**: 层关键字参数。

## add

```
keras.layers.add(inputs)
```

Add 层的函数式接口。

## 参数

- **inputs**: 一个输入张量的列表（列表大小至少为 2）。
- **\*\*kwargs**: 层关键字参数。

## 返回

一个张量，所有输入张量的和。

## 示例

```
import keras

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

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

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

## subtract

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

Subtract 层的函数式接口。

### 参数

- **inputs**: 一个列表的输入张量（列表大小准确为 2）。
- **\*\*kwargs**: 层的关键字参数。

### 返回

一个张量，两个输入张量的差。

### 示例

```
import keras

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

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

## multiply

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

Multiply 层的函数式接口。

### 参数

- **inputs**: 一个列表的输入张量（列表大小至少为 2）。

- **\*\*kwargs**: 层的关键字参数。

## 返回

一个张量，所有输入张量的逐元素乘积。

## average

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

Average 层的函数式接口。

## 参数

- **inputs**: 一个列表的输入张量（列表大小至少为 2）。
- **\*\*kwargs**: 层的关键字参数。

## 返回

一个张量，所有输入张量的平均值。

## maximum

```
keras.layers.maximum(inputs)
```

Maximum 层的函数式接口。

## 参数

- **inputs**: 一个列表的输入张量（列表大小至少为 2）。
- **\*\*kwargs**: 层的关键字参数。

## 返回

一个张量，所有张量的逐元素的最大值。

## minimum

```
keras.layers.minimum(inputs)
```

Minimum 层的函数式接口。

## 参数

- **inputs**: 一个列表的输入张量（列表大小至少为 2）。
- **\*\*kwargs**: 层的关键字参数。

## 返回

一个张量，所有张量的逐元素的最小值。

## concatenate

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

`Concatenate` 层的函数式接口。

## 参数

- **inputs**: 一个列表的输入张量（列表大小至少为 2）。
- **axis**: 串联的轴。
- **\*\*kwargs**: 层的关键字参数。

## 返回

一个张量，所有输入张量通过 `axis` 轴串联起来的输出张量。

## dot

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

`Dot` 层的函数式接口。

## 参数

- **inputs**: 一个列表的输入张量（列表大小至少为 2）。
- **axes**: 整数或者整数元组，一个或者几个进行点积的轴。
- **normalize**: 是否在点积之前对即将进行点积的轴进行 L2 标准化。如果设置成 True，那么输出两个样本之间的余弦相似值。
- **\*\*kwargs**: 层的关键字参数。

## 返回

一个张量，所有输入张量样本之间的点积。



# 高级激活层 Advanced Activations Layers

LeakyReLU

```
keras.layers.LeakyReLU(alpha=0.3)
```

带泄漏的 ReLU。

当神经元未激活时，它仍允许赋予一个很小的梯度：  $f(x) = \text{alpha} * x \text{ for } x < 0,$   
 $f(x) = x \text{ for } x \geq 0.$

## 输入尺寸

可以是任意的。如果将该层作为模型的第一层，则需要指定 `input_shape` 参数（整数元组，不包含样本数量的维度）。

## 输出尺寸

与输入相同。

## 参数

- `alpha`: float  $\geq 0$ 。负斜率系数。

## 参考文献

- [Rectifier Nonlinearities Improve Neural Network Acoustic Models](#)

PReLU

```
keras.layers.PReLU(alpha_initializer='zeros', alpha_regularizer=None,  
alpha_constraint=None, shared_axes=None)
```

参数化的 ReLU。

形式：`f(x) = alpha * x for x < 0, f(x) = x for x >= 0`, 其中 `alpha` 是一个可学习的数组，尺寸与 `x` 相同。

## 输入尺寸

可以是任意的。如果将这一层作为模型的第一层，则需要指定 `input_shape` 参数（整数元组，不包含样本数量的维度）。

## 输出尺寸

与输入相同。

## 参数

- `alpha_initializer`: 权重的初始化函数。
- `alpha_regularizer`: 权重的正则化方法。
- `alpha_constraint`: 权重的约束。
- `shared_axes`: 激活函数共享可学习参数的轴。例如，如果输入特征图来自输出形状为`(batch, height, width, channels)` 的 2D 卷积层，而且你希望跨空间共享参数，以便每个滤波器只有一组参数，可设置 `shared_axes=[1, 2]`。

## 参考文献

- [Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification](#)

ELU

```
keras.layers.ELU(alpha=1.0)
```

指数线性单元。

形式: `f(x) = alpha * (exp(x) - 1.) for x < 0, f(x) = x for x >= 0.`

## 输入尺寸

可以是任意的。如果将这一层作为模型的第一层，则需要指定 `input_shape` 参数（整数元组，不包含样本数量的维度）。

## 输出尺寸

与输入相同。

## 参数

- `alpha`: 负因子的尺度。

## 参考文献

- [Fast and Accurate Deep Network Learning by Exponential Linear Units \(ELUs\)](#)

ThresholdedReLU

```
keras.layers.ThresholdedReLU(theta=1.0)
```

带阈值的修正线性单元。

形式: `f(x) = x for x > theta, f(x) = 0 otherwise.`

## 输入尺寸

可以是任意的。如果将这一层作为模型的第一层，则需要指定 `input_shape` 参数（整数元组，不包含样本数量的维度）。

## 输出尺寸

与输入相同。

## 参数

- **theta**: float  $\geq 0$ 。激活的阈值位。

## 参考文献

- [Zero-Bias Autoencoders and the Benefits of Co-Adapting Features](#)

## Softmax

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

Softmax 激活函数。

## 输入尺寸

可以是任意的。如果将这一层作为模型的第一层，则需要指定 `input_shape` 参数（整数元组，不包含样本数量的维度）。

## 输出尺寸

与输入相同。

### 参数

- **axis**: 整数，应用 softmax 标准化的轴。

## ReLU

[\[source\]](#)

```
keras.layers.ReLU(max_value=None, negative_slope=0.0, threshold=0.0)
```

ReLU 激活函数。

使用默认值时，它返回逐个元素的 `max(x, 0)`。

否则：

- 如果 `x >= max_value`，返回 `f(x) = max_value`，
- 如果 `threshold <= x < max_value`，返回 `f(x) = x`，
- 否则，返回 `f(x) = negative_slope * (x - threshold)`。

## 输入尺寸

可以是任意的。如果将这一层作为模型的第一层，则需要指定 `input_shape` 参数（整数元组，不包含样本数量的维度）。

## 输出尺寸

与输入相同。

### 参数

- **max\_value**: 浮点数，最大的输出值。
- **negative\_slope**: float  $\geq 0$ . 负斜率系数。
- **threshold**: float。“thresholded activation”的阈值。



# 标准化层 Normalization Layers

## BatchNormalization

```
keras.layers.BatchNormalization(axis=-1, momentum=0.99,  
epsilon=0.001, center=True, scale=True, beta_initializer='zeros',  
gamma_initializer='ones', moving_mean_initializer='zeros',  
moving_variance_initializer='ones', beta_regularizer=None,  
gamma_regularizer=None, beta_constraint=None, gamma_constraint=None)
```

批量标准化层 (Ioffe and Szegedy, 2014)。

在每一个批次的数据中标准化前一层的激活项，即，应用一个维持激活项平均值接近 0，标准差接近 1 的转换。

### 参数

- **axis**: 整数，需要标准化的轴（通常是特征轴）。例如，在 `data_format="channels_first"` 的 `Conv2D` 层之后，在 `BatchNormalization` 中设置 `axis=1`。
- **momentum**: 移动均值和移动方差的动量。
- **epsilon**: 增加到方差的小的浮点数，以避免除以零。
- **center**: 如果为 `True`，把 `beta` 的偏移量加到标准化的张量上。如果为 `False`，`beta` 被忽略。
- **scale**: 如果为 `True`，乘以 `gamma`。如果为 `False`，`gamma` 不使用。当下一层为线性层（或者例如 `nn.relu`），这可以被禁用，因为缩放将由下一层完成。
- **beta\_initializer**: `beta` 权重的初始化方法。
- **gamma\_initializer**: `gamma` 权重的初始化方法。
- **moving\_mean\_initializer**: 移动均值的初始化方法。
- **moving\_variance\_initializer**: 移动方差的初始化方法。
- **beta\_regularizer**: 可选的 `beta` 权重的正则化方法。
- **gamma\_regularizer**: 可选的 `gamma` 权重的正则化方法。
- **beta\_constraint**: 可选的 `beta` 权重的约束方法。
- **gamma\_constraint**: 可选的 `gamma` 权重的约束方法。

### 输入尺寸

可以是任意的。如果将这一层作为模型的第一层，则需要指定 `input_shape` 参数（整数元组，不包含样本数量的维度）。

## 输出尺寸

与输入相同。

## 参考文献

- [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#)



噪声层 Noise layers

GaussianNoise

```
keras.layers.GaussianNoise(stddev)
```

应用以 0 为中心的加性高斯噪声。

这对缓解过拟合很有用（你可以将其视为随机数据增强的一种形式）。高斯噪声（GS）是对真实输入的腐蚀过程的自然选择。

由于它是一个正则化层，因此它只在训练时才被激活。

## 参数

- **stddev**: float，噪声分布的标准差。

## 输入尺寸

可以是任意的。如果将该层作为模型的第一层，则需要指定 `input_shape` 参数（整数元组，不包含样本数量的维度）。

## 输出尺寸

与输入相同。

## GaussianDropout

```
keras.layers.GaussianDropout(rate)
```

应用以 1 为中心的 乘性高斯噪声。

由于它是一个正则化层，因此它只在训练时才被激活。

### 参数

- **rate**: float，丢弃概率（与 Dropout 相同）。这个乘性噪声的标准差为  $\text{sqrt}(\text{rate} / (1 - \text{rate}))$ 。

### 输入尺寸

可以是任意的。如果将该层作为模型的第一层，则需要指定 `input_shape` 参数（整数元组，不包含样本数量的维度）。

## 输出尺寸

与输入相同。

## 参考文献

- [Dropout: A Simple Way to Prevent Neural Networks from Overfitting Srivastava, Hinton, et al. 2014](#)

## AlphaDropout

[\[source\]](#)

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

将 Alpha Dropout 应用到输入。

Alpha Dropout 是一种 Dropout，它保持输入的平均值和方差与原来的值不变，以确保即使在 dropout 后也能实现自我归一化。通过随机将激活设置为负饱和值，Alpha Dropout 非常适合按比例缩放的指数线性单元 (SELU)。

## 参数

- **rate**: float，丢弃概率（与 Dropout 相同）。这个乘性噪声的标准差为 `sqrt(rate / (1 - rate))`。
- **noise\_shape**: 一个类型为 `int32` 的 1D `Tensor`，表示随机生成 keep/drop 标识的尺寸。
- **seed**: 用作随机种子的 Python 整数。

## 输入尺寸

可以是任意的。如果将该层作为模型的第一层，则需要指定 `input_shape` 参数（整数元组，不包含样本数量的维度）。

## 输出尺寸

与输入相同。

## 参考文献

- [Self-Normalizing Neural Networks](#)



# 层封装器 Layer wrappers

## TimeDistributed

```
keras.layers.TimeDistributed(layer)
```

这个封装器将一个层应用于输入的每个时间片。

输入至少为 3D，且第一个维度应该是时间所表示的维度。

考虑 32 个样本的一个 batch，其中每个样本是 10 个 16 维向量的序列。那么这个 batch 的输入尺寸为 `(32, 10, 16)`，而 `input_shape` 不包含样本数量的维度，为 `(10, 16)`。

你可以使用 `TimeDistributed` 来将 `Dense` 层独立地应用到这 10 个时间步的每一个：

```
# 作为模型第一层
model = Sequential()
model.add(TimeDistributed(Dense(8), input_shape=(10, 16)))
# 现在 model.output_shape == (None, 10, 8)
```

输出的尺寸为 `(32, 10, 8)`。

在后续的层中，将不再需要 `input_shape`：

```
model.add(TimeDistributed(Dense(32)))
# 现在 model.output_shape == (None, 10, 32)
```

输出的尺寸为 `(32, 10, 32)`。

`TimeDistributed` 可以应用于任意层，不仅仅是 `Dense`，例如运用于 `Conv2D` 层：

```
model = Sequential()
model.add(TimeDistributed(Conv2D(64, (3, 3)),
    input_shape=(10, 299, 299, 3)))
```

## 参数

- **layer**: 一个网络层实例。

## Bidirectional

[\[source\]](#)

```
keras.layers.Bidirectional(layer, merge_mode='concat', weights=None)
```

RNN 的双向封装器，对序列进行前向和后向计算。

## 参数

- **layer**: Recurrent 实例。
- **merge\_mode**: 前向和后向 RNN 的输出的结合模式。为 {'sum', 'mul', 'concat', 'ave', None} 其中之一。如果是 None，输出不会被结合，而是作为一个列表被返回。
- **weights**: 双向模型中要加载的初始权重。

## 异常

- **ValueError**: 如果参数 merge\_mode 非法。

## 示例

```
model = Sequential()
model.add(Bidirectional(LSTM(10, return_sequences=True),
    input_shape=(5, 10)))
model.add(Bidirectional(LSTM(10)))
model.add(Dense(5))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')
```



# 编写你自己的 Keras 层

对于简单、无状态的自定义操作，你也许可以通过 `layers.core.Lambda` 层来实现。但是对于那些包含了可训练权重的自定义层，你应该自己实现这种层。

这是一个 **Keras 2.0** 中，Keras 层的骨架（如果你用的是旧的版本，请更新到新版）。你只需要实现三个方法即可：

- `build(input_shape)`：这是你定义权重的地方。这个方法必须设 `self.built = True`，可以通过调用 `super([Layer], self).build()` 完成。
- `call(x)`：这里是编写层的功能逻辑的地方。你只需要关注传入 `call` 的第一个参数：输入张量，除非你希望你的层支持masking。
- `compute_output_shape(input_shape)`：如果你的层更改了输入张量的形状，你应该在这里定义形状变化的逻辑，这让Keras能够自动推断各层的形状。

```
from keras import backend as K
from keras.engine.topology import Layer

class MyLayer(Layer):

    def __init__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(MyLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        # 为该层创建一个可训练的权重
        self.kernel = self.add_weight(name='kernel',
                                      shape=(input_shape[1], self.output_dim),
                                      initializer='uniform',
                                      trainable=True)
        super(MyLayer, self).build(input_shape) # 一定要在最后调用它

    def call(self, x):
        return K.dot(x, self.kernel)

    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.output_dim)
```

还可以定义具有多个输入张量和多个输出张量的 Keras 层。为此，你应该假设方法 `build(input_shape)`，`call(x)` 和 `compute_output_shape(input_shape)` 的输入输出都是列表。这里是一个例子，与上面那个相似：

```
from keras import backend as K
from keras.engine.topology import Layer

class MyLayer(Layer):
```

```
def __init__(self, output_dim, **kwargs):
    self.output_dim = output_dim
    super(MyLayer, self).__init__(**kwargs)

def build(self, input_shape):
    assert isinstance(input_shape, list)
    # 为该层创建一个可训练的权重
    self.kernel = self.add_weight(name='kernel',
                                  shape=(input_shape[0][1],
                                         self.output_dim),
                                  initializer='uniform',
                                  trainable=True)
    super(MyLayer, self).build(input_shape) # 一定要在最后调用它

def call(self, x):
    assert isinstance(x, list)
    a, b = x
    return [K.dot(a, self.kernel) + b, K.mean(b, axis=-1)]

def compute_output_shape(self, input_shape):
    assert isinstance(input_shape, list)
    shape_a, shape_b = input_shape
    return [(shape_a[0], self.output_dim), shape_b[:-1]]
```

已有的 Keras 层就是实现任何层的好例子。不要犹豫阅读源码！



# 序列预处理

## TimeseriesGenerator

```
keras.preprocessing.sequence.TimeseriesGenerator(data, targets,  
length, sampling_rate=1, stride=1, start_index=0, end_index=None,  
shuffle=False, reverse=False, batch_size=128)
```

用于生成批量时序数据的实用工具类。

这个类以一系列由相等间隔以及一些时间序列参数（例如步长、历史长度等）汇集的数据点作为输入，以生成用于训练/验证的批次数据。

### 参数

- **data**: 可索引的生成器（例如列表或 Numpy 数组），包含连续数据点（时间步）。数据应该是 2D 的，且第 0 个轴为时间维度。
- **targets**: 对应于 `data` 的时间步的目标值。它应该与 `data` 的长度相同。
- **length**: 输出序列的长度（以时间步数表示）。
- **sampling\_rate**: 序列内连续各个时间步之间的周期。对于周期 `r`, 时间步 `data[i]`, `data[i-r], ... data[i - length]` 被用于生成样本序列。
- **stride**: 连续输出序列之间的周期。对于周期 `s`, 连续输出样本将为 `data[i]`, `data[i+s]`, `data[i+2*s]` 等。
- **start\_index**: 在 `start_index` 之前的数据点在输出序列中将不被使用。这对保留部分数据以进行测试或验证很有用。
- **end\_index**: 在 `end_index` 之后的数据点在输出序列中将不被使用。这对保留部分数据以进行测试或验证很有用。
- **shuffle**: 是否打乱输出样本，还是按照时间顺序绘制它们。
- **reverse**: 布尔值: 如果 `true`, 每个输出样本中的时间步将按照时间倒序排列。
- **batch\_size**: 每个批次中的时间序列样本数（可能除最后一个外）。

### 返回

一个 [Sequence](#) 实例。

### 示例

```
from keras.preprocessing.sequence import TimeseriesGenerator  
import numpy as np
```

```

data = np.array([[i] for i in range(50)])
targets = np.array([[i] for i in range(50)])

data_gen = TimeseriesGenerator(data, targets,
                               length=10, sampling_rate=2,
                               batch_size=2)
assert len(data_gen) == 20

batch_0 = data_gen[0]
x, y = batch_0
assert np.array_equal(x,
                      np.array([[[0], [2], [4], [6], [8]],
                                [[1], [3], [5], [7], [9]]]))
assert np.array_equal(y,
                      np.array([[10], [11]]))

```

## pad\_sequences

```

keras.preprocessing.sequence.pad_sequences(sequences, maxlen=None,
                                         dtype='int32', padding='pre', truncating='pre', value=0.0)

```

将多个序列截断或补齐为相同长度。

该函数将一个 `num_samples` 的序列（整数列表）转化为一个 2D Numpy 矩阵，其尺寸为 `(num_samples, num_timesteps)`。`num_timesteps` 要么是给定的 `maxlen` 参数，要么是最长序列的长度。

比 `num_timesteps` 短的序列将在末端以 `value` 值补齐。

比 `num_timesteps` 长的序列将会被截断以满足所需要的长度。补齐或截断发生的位置分别由参数 `pading` 和 `truncating` 决定。

向前补齐为默认操作。

## 参数

- **sequences**: 列表的列表，每一个元素是一个序列。
- **maxlen**: 整数，所有序列的最大长度。
- **dtype**: 输出序列的类型。要使用可变长度字符串填充序列，可以使用 `object`。
- **padding**: 字符串，'pre' 或 'post'，在序列的前端补齐还是在后端补齐。
- **truncating**: 字符串，'pre' 或 'post'，移除长度大于 `maxlen` 的序列的值，要么在序列前端截断，要么在后端。
- **value**: 浮点数，表示用来补齐的值。

## 返回

- **x**: Numpy 矩阵，尺寸为 `(len(sequences), maxlen)`。

## 异常

- `ValueError`: 如果截断或补齐的值无效，或者序列条目的形状无效。

## skipgrams

```
keras.preprocessing.sequence.skipgrams(sequence, vocabulary_size,  
window_size=4, negative_samples=1.0, shuffle=True, categorical=False,  
sampling_table=None, seed=None)
```

生成 skipgram 词对。

该函数将一个单词索引序列（整数列表）转化为以下形式的单词元组：

- （单词, 同窗口的单词），标签为 1（正样本）。
- （单词, 来自词汇表的随机单词），标签为 0（负样本）。

若要了解更多和 Skipgram 有关的知识，请参阅这份由 Mikolov 等人发表的经典论文：[Efficient Estimation of Word Representations in Vector Space](#)

## 参数

- **sequence**: 一个编码为单词索引（整数）列表的词序列（句子）。如果使用一个 `sampling_table`，词索引应该以一个相关数据集的词的排名匹配（例如，10 将会编码为第 10 个最长出现的词）。注意词汇表中的索引 0 是非单词，将被跳过。
- **vocabulary\_size**: 整数，最大可能词索引 + 1
- **window\_size**: 整数，采样窗口大小（技术上是半个窗口）。词 `w_i` 的窗口是 `[i - window_size, i + window_size+1]`。
- **negative\_samples**: 大于等于 0 的浮点数。0 表示非负（即随机）采样。1 表示与正样本数相同。
- **shuffle**: 是否在返回之前将这些词语打乱。
- **categorical**: 布尔值。如果 `False`，标签将为整数（例如 `[0, 1, 1 ... ]`），如果 `True`，标签将为分类，例如 `[[1,0],[0,1],[0,1] ... ]`。
- **sampling\_table**: 尺寸为 `vocabulary_size` 的 1D 数组，其中第 `i` 项编码了排名第 `i` 的词的采样概率。
- **seed**: 随机种子。

## 返回

couples, labels: 其中 `couples` 是整数对, `labels` 是 0 或 1。

## 注意

按照惯例, 词汇表中的索引 0 是非单词, 将被跳过。

## make\_sampling\_table

```
keras.preprocessing.sequence.make_sampling_table(size, sampling_factor=1e-05)
```

生成一个基于单词的概率采样表。

用来生成 `skipgrams` 的 `sampling_table` 参数。`sampling_table[i]` 是数据集中第  $i$  个最常见词的采样概率 (出于平衡考虑, 出现更频繁的词应该被更少地采样)。

采样概率根据 word2vec 中使用的采样分布生成:

```
p(word) = (min(1, sqrt(word_frequency / sampling_factor)) /  
(word_frequency / sampling_factor)))
```

我们假设单词频率遵循 Zipf 定律 ( $s=1$ ) , 来导出 `frequency(rank)` 的数值近似:

$\text{frequency}(\text{rank}) \sim 1/(\text{rank} * (\log(\text{rank}) + \gamma) + 1/2 - 1/(12*\text{rank}))$ , 其中  $\gamma$  为 Euler-Mascheroni 常量。

## 参数

- **size**: 整数, 可能采样的单词数量。
- **sampling\_factor**: word2vec 公式中的采样因子。

## 返回

一个长度为 `size` 大小的 1D Numpy 数组, 其中第  $i$  项是排名为  $i$  的单词的采样概率。



# 文本预处理

## Text Preprocessing

### Tokenizer

[source]

```
keras.preprocessing.text.Tokenizer(num_words=None,
                                    filters='!"#$%&()*+,-./:;=>?@[ \\]^_`{|}'
~\t\n',
                                    lower=True,
                                    split=' ',
                                    char_level=False,
                                    oov_token=None,
                                    document_count=0)
```

文本标记实用类。

该类允许使用两种方法向量化一个文本语料库： 将每个文本转化为一个整数序列（每个整数都是词典中标记的索引）； 或者将其转化为一个向量， 其中每个标记的系数可以是二进制值、词频、TF-IDF 权重等。

### 参数

- **num\_words**: 需要保留的最大词数， 基于词频。 只有最常出现的 `num_words-1` 词会被保留。
- **filters**: 一个字符串， 其中每个元素是一个将从文本中过滤掉的字符。 默认值是所有标点符号， 加上制表符和换行符， 减去 ‘ ’ 字符。
- **lower**: 布尔值。 是否将文本转换为小写。
- **split**: 字符串。 按该字符串切割文本。
- **char\_level**: 如果为 True，则每个字符都将被视为标记。
- **oov\_token**: 如果给出， 它将被添加到 `word_index` 中，并用于在 `text_to_sequence` 调用期间替换词汇表外的单词。

默认情况下， 删除所有标点符号， 将文本转换为空格分隔的单词序列（单词可能包含 ‘ ’ 字符）。 这些序列然后被分割成标记列表。 然后它们将被索引或向量化。

0 是不会被分配给任何单词的保留索引。

### hashing\_trick

```
keras.preprocessing.text.hashing_trick(text,
                                       n,
                                       hash_function=None,
                                       filters='!"#$%&()*+,-./:;=>?@[\'\n\']^_`{|}~\t\n',
                                       lower=True,
                                       split=' ')
```

将文本转换为固定大小散列空间中的索引序列。

## 参数

- **text**: 输入文本（字符串）。
- **n**: 散列空间维度。
- **hash\_function**: 默认为 python 散列函数，可以是 'md5' 或任意接受输入字符串并返回整数的函数。注意 'hash' 不是稳定的散列函数，所以它在不同的运行中不一致，而 'md5' 是一个稳定的散列函数。
- **filters**: 要过滤的字符列表（或连接），如标点符号。默认：'!"#\$%&()\*+,-./:;=>?@[\'\n\']^\_`{|}~\t\n'，包含基本标点符号，制表符和换行符。
- **lower**: 布尔值。是否将文本转换为小写。
- **split**: 字符串。按该字符串切割文本。

## 返回

整数词索引列表（唯一性无法保证）。

0 是不会被分配给任何单词的保留索引。

由于哈希函数可能发生冲突，可能会将两个或更多字分配给同一索引。碰撞的概率与散列空间的维度和不同对象的数量有关。

## one\_hot

```
keras.preprocessing.text.one_hot(text,
                                 n,
                                 filters='!"#$%&()*+,-./:;=>?@[\'\n\']^_`{|}~\t\n',
                                 lower=True,
                                 split=' ')
```

One-hot 将文本编码为大小为 n 的单词索引列表。

这是 `hashing_trick` 函数的一个封装，使用 `hash` 作为散列函数；单词索引映射无保证唯一性。

## 参数

- **text**: 输入文本（字符串）。
- **n**: 整数。词汇表尺寸。
- **filters**: 要过滤的字符列表（或连接），如标点符号。默认：`!"#$%&()*+,-./:;<=>?\n@[\r\n]^_`{|}~\t\n`，包含基本标点符号，制表符和换行符。
- **lower**: 布尔值。是否将文本转换为小写。
- **split**: 字符串。按该字符串切割文本。

## 返回

[1, n] 之间的整数列表。每个整数编码一个词（唯一性无法保证）。

## text\_to\_word\_sequence

```
keras.preprocessing.text.text_to_word_sequence(text,
                                              filters='!"#$%&()*+,-./:;<=>?\n@[\r\n]^_`{|}~\t\n',
                                              lower=True,
                                              split=' ')
```

将文本转换为单词（或标记）的序列。

## 参数

- **text**: 输入文本（字符串）。
- **filters**: 要过滤的字符列表（或连接），如标点符号。默认：`!"#$%&()*+,-./:;<=>?\n@[\r\n]^_`{|}~\t\n`，包含基本标点符号，制表符和换行符。
- **lower**: 布尔值。是否将文本转换为小写。
- **split**: 字符串。按该字符串切割文本。

## 返回

词或标记的列表。



# 图像预处理

## ImageDataGenerator 类

[\[source\]](#)

```
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,
                                              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='channels_last',
                                              validation_split=0.0,
                                              interpolation_order=1,
                                              dtype='float32')
```

通过实时数据增强生成张量图像数据批次。数据将不断循环（按批次）。

### 参数

- **featurewise\_center**: 布尔值。将输入数据的均值设置为 0，逐特征进行。
- **samplewise\_center**: 布尔值。将每个样本的均值设置为 0。
- **featurewise\_std\_normalization**: Boolean. 布尔值。将输入除以数据标准差，逐特征进行。
- **samplewise\_std\_normalization**: 布尔值。将每个输入除以其标准差。
- **zca\_epsilon**: ZCA 白化的 epsilon 值，默认为 1e-6。
- **zca\_whitening**: 布尔值。是否应用 ZCA 白化。
- **rotation\_range**: 整数。随机旋转的度数范围。
- **width\_shift\_range**: 浮点数、一维数组或整数
  - float: 如果 <1，则是除以总宽度的值，或者如果 >=1，则为像素值。

- 1-D 数组: 数组中的随机元素。
- int: 来自间隔 `(-width_shift_range, +width_shift_range)` 之间的整数个像素。
- `width_shift_range=2` 时, 可能值是整数 `[-1, 0, +1]`, 与 `width_shift_range=[-1, 0, +1]` 相同; 而 `width_shift_range=1.0` 时, 可能值是 `[-1.0, +1.0)` 之间的浮点数。
- **height\_shift\_range**: 浮点数、一维数组或整数
  - float: 如果 `<1`, 则是除以总宽度的值, 或者如果 `>=1`, 则为像素值。
  - 1-D array-like: 数组中的随机元素。
  - int: 来自间隔 `(-height_shift_range, +height_shift_range)` 之间的整数个像素。
  - `height_shift_range=2` 时, 可能值是整数 `[-1, 0, +1]`, 与 `height_shift_range=[-1, 0, +1]` 相同; 而 `height_shift_range=1.0` 时, 可能值是 `[-1.0, +1.0)` 之间的浮点数。
- **brightness\_range**: 两个浮点数的元组或列表。从中选择亮度偏移值的范围。
- **shear\_range**: 浮点数。剪切强度 (以弧度逆时针方向剪切角度)。
- **zoom\_range**: 浮点数 或 `[lower, upper]`。随机缩放范围。如果是浮点数, `[lower, upper] = [1-zoom_range, 1+zoom_range]`。
- **channel\_shift\_range**: 浮点数。随机通道转换的范围。
- **fill\_mode**: {"constant", "nearest", "reflect" or "wrap"} 之一。默认为 'nearest'。输入边界以外的点根据给定的模式填充:
  - 'constant': kkkkkkkk|abcd|kkkkkkkk (cval=k)
  - 'nearest': aaaaaaaaa|abcd|ddddddddd
  - 'reflect': abcdccba|abcd|dcbaabcd
  - 'wrap': abcdabcd|abcd|abcdabcd
- **cval**: 浮点数或整数。用于边界之外的点的值, 当 `fill_mode = "constant"` 时。
- **horizontal\_flip**: 布尔值。随机水平翻转。
- **vertical\_flip**: 布尔值。随机垂直翻转。
- **rescale**: 重缩放因子。默认为 None。如果是 None 或 0, 不进行缩放, 否则将数据乘以所提供的值 (在应用任何其他转换之前)。
- **preprocessing\_function**: 应用于每个输入的函数。这个函数会在任何其他改变之前运行。这个函数需要一个参数: 一张图像 (秩为 3 的 Numpy 张量), 并且应该输出一个同尺寸的 Numpy 张量。
- **data\_format**: 图像数据格式, {"channels\_first", "channels\_last"} 之一。"channels\_last" 模式表示图像输入尺寸应该为 `(samples, height, width, channels)`, "channels\_first" 模式表示输入尺寸应该为 `(samples, channels, height, width)`。默认为 在 Keras 配置文件

`~/.keras/keras.json` 中的 `image_data_format` 值。如果你从未设置它，那它就是“channels\_last”。

- **validation\_split**: 浮点数。Float. 保留用于验证的图像的比例（严格在0和1之间）。
- **dtype**: 生成数组使用的数据类型。

## 示例

使用 `.flow(x, y)` 的例子：

```
(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)

# 计算特征归一化所需的数量
# (如果应用 ZCA 白化，将计算标准差，均值，主成分)
datagen.fit(x_train)

# 使用实时数据增益的批数据对模型进行拟合：
model.fit_generator(datagen.flow(x_train, y_train, batch_size=32),
                     steps_per_epoch=len(x_train) / 32, epochs=epochs)

# 这里有一个更「手动」的例子
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:
            # 我们需要手动打破循环，
            # 因为生成器会无限循环
            break
```

使用 `.flow_from_directory(directory)` 的例子：

```
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)

```

同时转换图像和蒙版 (mask) 的例子。

```

# 创建两个相同参数的实例
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)

# 为 fit 和 flow 函数提供相同的种子和关键字参数
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)

# 将生成器组合成一个产生图像和蒙版 (mask) 的生成器
train_generator = zip(image_generator, mask_generator)

model.fit_generator(
    train_generator,
    steps_per_epoch=2000,
    epochs=50)

```

使用 `.flow_from_dataframe(dataframe, directory)` 的例子:

```
train_df = pandas.read_csv("./train.csv")
valid_df = pandas.read_csv("./valid.csv")

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_dataframe(
    dataframe=train_df,
    directory='data/train',
    x_col="filename",
    y_col="class",
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary')

validation_generator = test_datagen.flow_from_dataframe(
    dataframe=valid_df,
    directory='data/validation',
    x_col="filename",
    y_col="class",
    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)
```

## ImageDataGenerator 类方法

### apply\_transform

```
apply_transform(x, transform_parameters)
```

根据给定的参数将变换应用于图像。

#### 参数

- **x**: 3D 张量，单张图像。

• **transform\_parameters**: 字符串 - 参数对表示的字典，用于描述转换。目前，使用字典中的以下参数：

- 'theta': 浮点数。旋转角度（度）。
- 'tx': 浮点数。在 x 方向上移动。
- 'ty': 浮点数。在 y 方向上移动。
- 'shear': 浮点数。剪切角度（度）。
- 'zx': 浮点数。放大 x 方向。
- 'zy': 浮点数。放大 y 方向。
- 'flip\_horizontal': 布尔值。水平翻转。
- 'flip\_vertical': 布尔值。垂直翻转。
- 'channel\_shift\_intensity': 浮点数。频道转换强度。
- 'brightness': 浮点数。亮度转换强度。

## 返回

输入的转换后版本（相同尺寸）。

## fit

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

将数据生成器用于某些示例数据。

它基于一组样本数据，计算与数据转换相关的内部数据统计。

当且仅当 `featurewise_center` 或 `featurewise_std_normalization` 或 `zca_whitening` 设置为 `True` 时才需要。

## 参数

- **x**: 样本数据。秩应该为 4。对于灰度数据，通道轴的值应该为 1；对于 RGB 数据，值应该为 3；对于 RGBA 数据，值应该为 4。
- **augment**: 布尔值（默认为 `False`）。是否使用随机样本扩张。
- **rounds**: 整数（默认为 1）。如果数据数据增强（`augment=True`），表明在数据上进行多少次增强。
- **seed**: 整数（默认 `None`）。随机种子。

## flow

```
flow(x, y=None, batch_size=32, shuffle=True, sample_weight=None, seed=None,
      save_to_dir=None, save_prefix='', save_format='png', subset=None)
```

采集数据和标签数组，生成批量增强数据。

### 参数

- **x**: 输入数据。秩为 4 的 Numpy 矩阵或元组。如果是元组，第一个元素应该包含图像，第二个元素是另一个 Numpy 数组或一列 Numpy 数组，它们不经过任何修改就传递给输出。可用于将模型杂项数据与图像一起输入。对于灰度数据，图像数组的通道轴的值应该为 1；对于 RGB 数据，其值应该为 3；对于 RGBA 数据，值应该为 4。
- **y**: 标签。
- **batch\_size**: 整数 (默认为 32)。
- **shuffle**: 布尔值 (默认为 True)。
- **sample\_weight**: 样本权重。
- **seed**: 整数 (默认为 None)。
- **save\_to\_dir**: None 或字符串 (默认为 None)。这使您可以选择指定要保存的正在生成的增强图片的目录 (用于可视化您正在执行的操作)。
- **save\_prefix**: 字符串 (默认 '')。保存图片的文件名前缀 (仅当 `save_to_dir` 设置时可用)。
- **save\_format**: "png", "jpeg" 之一 (仅当 `save_to_dir` 设置时可用)。默认: "png"。
- **subset**: 数据子集 ("training" 或 "validation")，如果在 `ImageDataGenerator` 中设置了 `validation_split`。

### 返回

一个生成元组 (`x`, `y`) 的 `Iterator`，其中 `x` 是图像数据的 Numpy 数组 (在单张图像输入时)，或 Numpy 数组列表 (在额外多个输入时)，`y` 是对应的标签的 Numpy 数组。如果 '`sample_weight`' 不是 `None`，生成的元组形式为 (`x`, `y`, `sample_weight`)。如果 `y` 是 `None`，只有 Numpy 数组 `x` 被返回。

## flow\_from\_dataframe

```
flow_from_dataframe(dataframe, directory=None, x_col='filename',
                    y_col='class', weight_col=None, 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', subset=None,
                    interpolation='nearest', validate_filenames=True)
```

输入 `dataframe` 和目录的路径，并生成批量的增强/标准化的数据。

这里有一个简单的教程：[http://bit.ly/keras\\_flow\\_from\\_dataframe](http://bit.ly/keras_flow_from_dataframe)

## 参数

- **dataframe**: Pandas `dataframe`, 其中一列字符串包含对应目录（或绝对路径，如果 `directory` 为 `None`）的图片文件路径。它应该根据 `class_mode` 来包含其他列：
  - 如果 `class_mode` 是 `"categorical"` (默认值), 它必须包含 `y_col` 列表示每张图片的类别。这一列的值可以是字符串/列表/元组，如果是一个单独的类，或者是列表/元组，如果是多个类。
  - 如果 `class_mode` 是 `"binary"` 或 `"sparse"`，它必须包含给定的 `y_col` 列表示每张图片的字符串类别。
  - 如果 `class_mode` 是 `"raw"` 或 `"multi_output"`，它必须包含 `y_col` 中指定的列。
  - 如果 `class_mode` 是 `"input"` 或 `None`，则不需要额外的列。
- **directory**: 字符串，读取图片的目录的路径，如果是 `None`, `x_col` 列中的数据必须是绝对路径。
- **x\_col**: 字符串, `dataframe` 中包含文件名列（或者绝对路径，如果 `directory` 是 `None`）。
- **y\_col**: 字符串或字符串列表, `dataframe` 中将作为目标数据的列。
- **weight\_col**: 字符串, `dataframe` 中包含样本权重的列。默认为 `None`。
- **target\_size**: 整数元组 (`height, width`)，默认为 `(256, 256)`。所有找到的图都会调整到这个维度。
- **color\_mode**: `"grayscale"`, `"rgb"`, `"rgba"` 之一。默认: `"rgb"`。图像是否转换为 1 个或 3 个颜色通道。
- **classes**: 可选的类别列表 (例如, `['dogs', 'cats']`)。默认: `None`。如未提供，类比列表将自动从 `y_col` 中推理出来, `y_col` 将会被映射为类别索引)。包含从类名到类索引的映射的字典可以通过属性 `class_indices` 获得。
- **class\_mode**: `"binary"`, `"categorical"`, `"input"`, `"multi_output"`, `"raw"`, `sparse"` 或 `None` 之一。默认: `"categorical"`。决定返回标签数组的类型：
  - `"binary"` : 1D numpy 数组二进制标签；
  - `"categorical"` : 2D numpy 数组 one-hot 编码标签，支持多标签输出；
  - `"input"` : 与输入图像相同的图像（主要用于与自动编码器一起使用）；
  - `"multi_output"` : 不同列的值的列表；
  - `"raw"` : `y_col` 列中值的 numpy 数组；
  - `"sparse"` : 1D numpy 数组整数标签；

- "other" 将是 `y_col` 数据的 numpy 数组；
- `None`：不返回任何标签（生成器只会产生批量的图像数据，这对使用 `model.predict_generator()`, `model.evaluate_generator()` 等很有用）。
- **batch\_size**: 批量数据的尺寸（默认：32）。
- **shuffle**: 是否混洗数据（默认：True）
- **seed**: 可选的混洗和转换的随即种子。
- **save\_to\_dir**: `None` 或 `str` (默认: `None`)。这允许你可选地指定要保存正在生成的增强图片的目录（用于可视化您正在执行的操作）。
- **save\_prefix**: 字符串。保存图片的文件名前缀（仅当 `save_to_dir` 设置时可用）。
- **save\_format**: "png", "jpeg" 之一（仅当 `save_to_dir` 设置时可用）。默认："png"。
- **follow\_links**: 是否跟随类子目录中的符号链接（默认：False）。
- **subset**: 数据子集 ("training" 或 "validation")，如果在 `ImageDataGenerator` 中设置了 `validation_split`。
- **interpolation**: 在目标大小与加载图像的大小不同时，用于重新采样图像的插值方法。支持的方法有 "nearest", "bilinear", and "bicubic"。如果安装了 1.1.3 以上版本的 PIL 的话，同样支持 "lanczos"。如果安装了 3.4.0 以上版本的 PIL 的话，同样支持 "box" 和 "hamming"。默认情况下，使用 "nearest"。
- **validate\_filenames**: 布尔值，是否验证 `x_col` 中的图片路径。如果 True，将忽略无效的图片。禁用这一选项会加速这一函数的执行。默认：True。

## Returns

一个生成 (`x`, `y`) 元组的 `DataFrameIterator`，其中 `x` 是一个包含一批尺寸为 (`batch_size`, \*`target_size`, `channels`) 的图像样本的 numpy 数组，`y` 是对应的标签的 numpy 数组。

## 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')
```

## 参数

- **directory**: 字符串，目标目录的路径。每个类应该包含一个子目录。任何在子目录树下的 PNG, JPG, BMP, PPM 或 TIF 图像，都将被包含在生成器中。更多细节，详见[此脚本](#)。
- **target\_size**: 整数元组 (`height`, `width`)，默认：(256, 256)。所有的图像将被调整到的尺寸。

- **color\_mode**: “grayscale”, “rgb”, “rgba” 之一。默认: “rgb”。图像是否被转换成 1, 3 或 4 个颜色通道。
- **classes**: 可选的类的子目录列表 (例如 `['dogs', 'cats']`)。默认: `None`。如果未提供, 类的列表将自动从 `directory` 下的 子目录名称/结构 中推断出来, 其中每个子目录都将被作为不同的类 (类名将按字典序映射到标签的索引)。包含从类名到类索引的映射的字典可以通过 `class_indices` 属性获得。
- **class\_mode**: “categorical”, “binary”, “sparse”, “input” 或 `None` 之一。默认: “categorical”。决定返回的标签数组的类型:
  - “categorical” 将是 2D one-hot 编码标签,
  - “binary” 将是 1D 二进制标签, “sparse” 将是 1D 整数标签,
  - “input” 将是与输入图像相同的图像 (主要用于自动编码器)。
  - 如果为 `None`, 不返回标签 (生成器将只产生批量的图像数据, 对于 `model.predict_generator()`, `model.evaluate_generator()` 等很有用)。请注意, 如果 `class_mode` 为 `None`, 那么数据仍然需要驻留在 `directory` 的子目录中才能正常工作。
- **batch\_size**: 一批数据的大小 (默认 32)。
- **shuffle**: 是否混洗数据 (默认 `True`)。
- **seed**: 可选随机种子, 用于混洗和转换。
- **save\_to\_dir**: `None` 或 字符串 (默认 `None`)。这使你可以最佳地指定正在生成的增强图片要保存的目录 (用于可视化你在做什么)。
- **save\_prefix**: 字符串。保存图片的文件名前缀 (仅当 `save_to_dir` 设置时可用)。
- **save\_format**: “png”, “jpeg” 之一 (仅当 `save_to_dir` 设置时可用)。默认: “png”。
- **follow\_links**: 是否跟踪类子目录中的符号链接 (默认为 `False`)。
- **subset**: 数据子集 (“training” 或 “validation”), 如果在 `ImageDataGenerator` 中设置了 `validation_split`。
- **interpolation**: 在目标大小与加载图像的大小不同时, 用于重新采样图像的插值方法。支持的方法有 `“nearest”`, `“bilinear”`, and `“bicubic”`。如果安装了 1.1.3 以上版本的 PIL 的话, 同样支持 `“lanczos”`。如果安装了 3.4.0 以上版本的 PIL 的话, 同样支持 `“box”` 和 `“hamming”`。默认情况下, 使用 `“nearest”`。

## 返回

一个生成 `(x, y)` 元组的 `DirectoryIterator`, 其中 `x` 是一个包含一批尺寸为 `(batch_size, *target_size, channels)` 的图像的 Numpy 数组, `y` 是对应标签的 Numpy 数组。

## get\_random\_transform

```
get_random_transform(img_shape, seed=None)
```

为转换生成随机参数。

### 参数

- **seed**: 随机种子
- **img\_shape**: 整数元组。被转换的图像的尺寸。

### 返回

包含随机选择的描述变换的参数的字典。

## random\_transform

```
random_transform(x, seed=None)
```

将随机变换应用于图像。

### 参数

- **x**: 3D 张量，单张图像。
- **seed**: 随机种子。

### 返回

输入的随机转换版本（相同形状）。

## standardize

```
standardize(x)
```

将标准化配置应用于一批输入。

由于该函数主要在内部用于对图像进行标准化处理并将其馈送到网络，所以 `x` 会就地更改。如果要创建 `x` 的副本，则会带来很大的性能成本。如果要应用此方法而不更改就地输入，则可以在这之前调用创建副本的方法：

```
standarize(np.copy(x))
```

## 参数

- **x**: 需要标准化的一批输入。

## 返回

标准化后的输入。



# 损失函数 Losses

## 损失函数的使用

损失函数（或称目标函数、优化评分函数）是编译模型时所需的两个参数之一：

```
model.compile(loss='mean_squared_error', optimizer='sgd')  
  
from keras import losses  
  
model.compile(loss=losses.mean_squared_error, optimizer='sgd')
```

你可以传递一个现有的损失函数名，或者一个 TensorFlow/Theano 符号函数。该符号函数为每个数据点返回一个标量，有以下两个参数：

- **y\_true**: 真实标签。TensorFlow/Theano 张量。
- **y\_pred**: 预测值。TensorFlow/Theano 张量，其 shape 与 y\_true 相同。

实际的优化目标是所有数据点的输出数组的平均值。

有关这些函数的几个例子，请查看 [losses source](#)。

## 可用损失函数

### mean\_squared\_error

```
keras.losses.mean_squared_error(y_true, y_pred)
```

### mean\_absolute\_error

```
keras.losses.mean_absolute_error(y_true, y_pred)
```

### mean\_absolute\_percentage\_error

```
keras.losses.mean_absolute_percentage_error(y_true, y_pred)
```

## mean\_squared\_logarithmic\_error

```
keras.losses.mean_squared_logarithmic_error(y_true, y_pred)
```

## squared\_hinge

```
keras.losses.squared_hinge(y_true, y_pred)
```

## hinge

```
keras.losses.hinge(y_true, y_pred)
```

## categorical\_hinge

```
keras.losses.categorical_hinge(y_true, y_pred)
```

## logcosh

```
keras.losses.logcosh(y_true, y_pred)
```

预测误差的双曲余弦的对数。

对于小的 `x`，`log(cosh(x))` 近似等于 `(x ** 2) / 2`。对于大的 `x`，近似于 `abs(x) - log(2)`。这表示 'logcosh' 与均方误差大致相同，但是不会受到偶尔疯狂的错误预测的强烈影响。

### 参数

- **y\_true**: 目标真实值的张量。
- **y\_pred**: 目标预测值的张量。

### 返回

每个样本都有一个标量损失的张量。

## huber\_loss

```
keras.losses.huber_loss(y_true, y_pred, delta=1.0)
```

## categorical\_crossentropy

```
keras.losses.categorical_crossentropy(y_true, y_pred, from_logits=False,
label_smoothing=0)
```

## sparse\_categorical\_crossentropy

```
keras.losses.sparse_categorical_crossentropy(y_true, y_pred,
from_logits=False, axis=-1)
```

## binary\_crossentropy

```
keras.losses.binary_crossentropy(y_true, y_pred, from_logits=False,
label_smoothing=0)
```

## kullback\_leibler\_divergence

```
keras.losses.kullback_leibler_divergence(y_true, y_pred)
```

## poisson

```
keras.losses.poisson(y_true, y_pred)
```

## cosine\_proximity

```
keras.losses.cosine_proximity(y_true, y_pred, axis=-1)
```

## is\_categorical\_crossentropy

```
keras.losses.is_categorical_crossentropy(loss)
```

**注意:** 当使用 `categorical_crossentropy` 损失时，你的目标值应该是分类格式 (即，如果你有 10 个类，每个样本的目标值应该是一个 10 维的向量，这个向量除了表示类别的那个索引为 1，其他均为 0)。为了将 整数目标值 转换为 分类目标值，你可以使用 Keras 实用函数 `to_categorical`：

```
from keras.utils.np_utils import to_categorical  
  
categorical_labels = to_categorical(int_labels, num_classes=None)
```

当使用 `sparse_categorical_crossentropy` 损失时，你的目标应该是整数。如果你是类别目标，应该使用 `categorical_crossentropy`。

`categorical_crossentropy` 是多类对数损失的另一种形式。



# 评估标准 Metrics

## 评价函数的用法

评价函数用于评估当前训练模型的性能。当模型编译后（compile），评价函数应该作为 metrics 的参数来输入。

```
model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=['mae', 'acc'])
```

```
from keras import metrics

model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=[metrics.mae, metrics.categorical_accuracy])
```

评价函数和 [损失函数](#) 相似，只不过评价函数的结果不会用于训练过程中。

我们可以传递已有的评价函数名称，或者传递一个自定义的 Theano/TensorFlow 函数来使用（查阅[自定义评价函数](#)）。

### 参数

- **y\_true**: 真实标签，Theano/Tensorflow 张量。
- **y\_pred**: 预测值。和 y\_true 相同尺寸的 Theano/TensorFlow 张量。

### 返回

返回一个表示全部数据点平均值的张量。

## 可使用的评价函数

### accuracy

```
keras.metrics.accuracy(y_true, y_pred)
```

### binary\_accuracy

```
keras.metrics.binary_accuracy(y_true, y_pred, threshold=0.5)
```

## categorical\_accuracy

```
keras.metrics.categorical_accuracy(y_true, y_pred)
```

## sparse\_categorical\_accuracy

```
keras.metrics.sparse_categorical_accuracy(y_true, y_pred)
```

## top\_k\_categorical\_accuracy

```
keras.metrics.top_k_categorical_accuracy(y_true, y_pred, k=5)
```

## sparse\_top\_k\_categorical\_accuracy

```
keras.metrics.sparse_top_k_categorical_accuracy(y_true, y_pred, k=5)
```

## cosine\_proximity

```
keras.metrics.cosine_proximity(y_true, y_pred, axis=-1)
```

## clone\_metric

```
keras.metrics.clone_metric(metric)
```

若有状态，返回评估指标的克隆，否则返回其本身。

## clone\_metrics

```
keras.metrics.clone_metrics(metrics)
```

克隆给定的评估指标序列/字典。

除以上评估指标，你还可以使用在损失函数页描述的损失函数作为评估指标。

## 自定义评价函数

自定义评价函数应该在编译的时候 (compile) 传递进去。该函数需要以 (`y_true`, `y_pred`) 作为输入参数，并返回一个张量作为输出结果。

```
import keras.backend as K

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

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



# 优化器 Optimizers

## 优化器的用法

优化器 (optimizer) 是编译 Keras 模型的所需的两个参数之一：

```
from keras import optimizers

model = Sequential()
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(Activation('softmax'))

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

你可以先实例化一个优化器对象，然后将它传入 `model.compile()`，像上述示例中一样，或者你可以通过名称来调用优化器。在后一种情况下，将使用优化器的默认参数。

```
# 传入优化器名称：默认参数将被采用  
model.compile(loss='mean_squared_error', optimizer='sgd')
```

## Keras 优化器的公共参数

参数 `clipnorm` 和 `clipvalue` 能在所有的优化器中使用，用于控制梯度裁剪（Gradient Clipping）：

```
from keras import optimizers

# 所有参数梯度将被裁剪，让其 l2 范数最大为 1: g * 1 / max(1, l2_norm)
sgd = optimizers.SGD(lr=0.01, clipnorm=1.)
```



```
from keras import optimizers

# 所有参数 d 梯度将被裁剪到数值范围内:
# 最大值 0.5
# 最小值 -0.5
sgd = optimizers.SGD(lr=0.01, clipvalue=0.5)
```



SGD

```
keras.optimizers.SGD(learning_rate=0.01, momentum=0.0, nesterov=False)
```

随机梯度下降优化器。

包含扩展功能的支持：

- 动量 (momentum) 优化,
- 学习率衰减 (每次参数更新后)
- Nesterov 动量 (NAG) 优化

## 参数

- **learning\_rate**: float  $\geq 0$ . 学习率。
- **momentum**: float  $\geq 0$ . 参数，用于加速 SGD 在相关方向上前进，并抑制震荡。
- **nesterov**: boolean. 是否使用 Nesterov 动量。

RMSprop

```
keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9)
```

RMSProp 优化器。

建议使用优化器的默认参数（除了学习率，它可以被自由调节）

这个优化器通常是训练循环神经网络 RNN 的不错选择。

## 参数

- **learning\_rate**: float  $\geq 0$ . 学习率。
- **rho**: float  $\geq 0$ . RMSProp 梯度平方的移动均值的衰减率。

## 参考文献

- [rmsprop: Divide the gradient by a running average of its recent magnitude](#)

Adagrad

```
keras.optimizers.Adagrad(learning_rate=0.01)
```

Adagrad 优化器。

Adagrad 是一种具有特定参数学习率的优化器，它根据参数在训练期间的更新频率进行自适应调整。参数接收的更新越多，更新越小。

建议使用优化器的默认参数。

## 参数

- **learning\_rate**: float  $\geq 0$ . 学习率。

## 参考文献

- [Adaptive Subgradient Methods for Online Learning and Stochastic Optimization](#)

Adadelta

```
keras.optimizers.Adadelta(learning_rate=1.0, rho=0.95)
```

Adadelta 优化器。

Adadelta 是 Adagrad 的一个具有更强鲁棒性的的扩展版本，它不是累积所有过去的梯度，而是根据渐变更新的移动窗口调整学习速率。这样，即使进行了许多更新，Adadelta 仍在继续学习。与 Adagrad 相比，在 Adadelta 的原始版本中，您无需设置初始学习率。在此版本中，与大多数其他 Keras 优化器一样，可以设置初始学习速率和衰减因子。

建议使用优化器的默认参数。

## 参数

- **learning\_rate**: float  $\geq 0$ . 初始学习率，默认为 1。建议保留默认值。
- **rho**: float  $\geq 0$ . Adadelta 梯度平方移动均值的衰减率。

## 参考文献

- [Adadelta - an adaptive learning rate method](#)

Adam

```
keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999,  
amsgrad=False)
```

Adam 优化器。

默认参数遵循原论文中提供的值。

## 参数

- **learning\_rate**: float  $\geq 0$ . 学习率。
- **beta\_1**: float,  $0 < \text{beta} < 1$ . 通常接近于 1。
- **beta\_2**: float,  $0 < \text{beta} < 1$ . 通常接近于 1。
- **amsgrad**: boolean. 是否应用此算法的 AMSGrad 变种，来自论文 “On the Convergence of Adam and Beyond”。

## 参考文献

- [Adam - A Method for Stochastic Optimization](#)
- [On the Convergence of Adam and Beyond](#)

## Adamax

```
keras.optimizers.Adamax(learning_rate=0.002, beta_1=0.9, beta_2=0.999)
```

Adamax 优化器，来自 Adam 论文的第七小节。

它是 Adam 算法基于无穷范数 (infinity norm) 的变种。默认参数遵循论文中提供的值。

## 参数

- **learning\_rate**: float  $\geq 0$ . 学习率。
- **beta\_1**: floats,  $0 < \text{beta} < 1$ . 通常接近于 1。
- **beta\_2**: floats,  $0 < \text{beta} < 1$ . 通常接近于 1。

## 参考文献

- [Adam - A Method for Stochastic Optimization](#)

## Nadam

[\[source\]](#)

```
keras.optimizers.Nadam(learning_rate=0.002, beta_1=0.9, beta_2=0.999)
```

Nesterov 版本 Adam 优化器。

正像 Adam 本质上是 RMSProp 与动量 momentum 的结合，Nadam 是采用 Nesterov momentum 版本的 Adam 优化器。

默认参数遵循论文中提供的值。建议使用优化器的默认参数。

## 参数

- **learning\_rate**: float  $\geq 0$ . 学习率。
- **beta\_1**: floats,  $0 < \text{beta} < 1$ . 通常接近于 1。
- **beta\_2**: floats,  $0 < \text{beta} < 1$ . 通常接近于 1。

## 参考文献

- [Nadam report](#)
- [On the importance of initialization and momentum in deep learning](#)



# 激活函数 Activations

## 激活函数的用法

激活函数可以通过设置单独的 `Activation` 层实现，也可以在构造层对象时通过传递 `activation` 参数实现：

```
from keras.layers import Activation, Dense  
  
model.add(Dense(64))  
model.add(Activation('tanh'))
```

等价于：

```
model.add(Dense(64, activation='tanh'))
```

你也可以通过传递一个逐元素运算的 Theano/TensorFlow/CNTK 函数来作为激活函数：

```
from keras import backend as K  
  
model.add(Dense(64, activation=K.tanh))  
model.add(Activation(K.tanh))
```

## 预定义激活函数

### elu

```
keras.activations.elu(x, alpha=1.0)
```

指数线性单元。

#### 参数

- `x`: 输入张量。
- `alpha`: 一个标量，表示负数部分的斜率。

#### 返回

线性指数激活：如果 `x > 0`，返回值为 `x`；如果 `x < 0` 返回值为 `alpha * (exp(x)-1)`

#### 参考文献

- [Fast and Accurate Deep Network Learning by Exponential Linear Units \(ELUs\)](#)

## softmax

```
keras.activations.softmax(x, axis=-1)
```

Softmax 激活函数。

### 参数

- **x**: 输入张量。
- **axis**: 整数，代表 softmax 所作用的维度。

### 返回

softmax 变换后的张量。

### 异常

- **ValueError**: 如果 `dim(x) == 1`。

## selu

```
keras.activations.selu(x)
```

可伸缩的指数线性单元 (SELU)。

SELU 等同于: `scale * elu(x, alpha)`，其中 `alpha` 和 `scale` 是预定义的常量。只要正确初始化权重（参见 `lecun_normal` 初始化方法）并且输入的数量「足够大」（参见参考文献获得更多信息），选择合适的 `alpha` 和 `scale` 的值，就可以在两个连续层之间保留输入的均值和方差。

### 参数

- **x**: 一个用来用于计算激活函数的张量或变量。

### 返回

可伸缩的指数线性激活: `scale * elu(x, alpha)`。

### 注意

- 与「`lecun_normal`」初始化方法一起使用。
- 与 `dropout` 的变种「`AlphaDropout`」一起使用。

## 参考文献

- [Self-Normalizing Neural Networks](#)

## softplus

```
keras.activations.softplus(x)
```

Softplus 激活函数。

### 参数

- **x**: 输入张量。

### 返回

Softplus 激活:  $\log(\exp(x) + 1)$ 。

## softsign

```
keras.activations.softsign(x)
```

Softsign 激活函数。

### 参数

- **x**: 输入张量。

### 返回

Softsign 激活:  $x / (\text{abs}(x) + 1)$ 。

## relu

```
keras.activations.relu(x, alpha=0.0, max_value=None, threshold=0.0)
```

整流线性单元。

使用默认值时，它返回逐元素的  $\max(x, 0)$ 。

否则，它遵循：

- 如果  $x \geq \text{max\_value}$  :  $f(x) = \text{max\_value}$  ,

- 如果 `threshold <= x < max_value`:  $f(x) = x$ ,
- 否则:  $f(x) = \text{alpha} * (x - \text{threshold})$ 。

## 参数

- **x**: 输入张量。
- **alpha**: 负数部分的斜率。默认为 0。
- **max\_value**: 输出的最大值。
- **threshold**: 浮点数。Thresholded activation 的阈值值。

## 返回

一个张量。

## tanh

```
keras.activations.tanh(x)
```

双曲正切激活函数。

## 参数

- **x**: 输入张量。

## 返回

双曲正切激活函数:  $\tanh(x) = (\exp(x) - \exp(-x)) / (\exp(x) + \exp(-x))$

## sigmoid

```
sigmoid(x)
```

Sigmoid 激活函数。

## 参数

- **x**: 输入张量。

## 返回

Sigmoid激活函数:  $1 / (1 + \exp(-x))$ 。

## hard\_sigmoid

```
hard_sigmoid(x)
```

Hard sigmoid 激活函数。

计算速度比 sigmoid 激活函数更快。

### 参数

- **x**: 输入张量。

### 返回

Hard sigmoid 激活函数：

- 如果  $x < -2.5$ ，返回 0。
- 如果  $x > 2.5$ ，返回 1。
- 如果  $-2.5 \leq x \leq 2.5$ ，返回  $0.2 * x + 0.5$ 。

## exponential

```
keras.activations.exponential(x)
```

自然数指数激活函数。

## linear

```
keras.activations.linear(x)
```

线性激活函数（即不做任何改变）

### 参数

- **x**: 输入张量。

### 返回

输入张量，不变。

## 高级激活函数

对于 Theano/TensorFlow/CNTK 不能表达的复杂激活函数，如含有可学习参数的激活函数，可通过[高级激活函数](#)实现，可以在 `keras.layers.advanced_activations` 模块中找到。这些高级激活函数包括 PReLU 和 LeakyReLU。



## 回调函数使用

回调函数是一个函数的合集，会在训练的阶段中所使用。你可以使用回调函数来查看训练模型的内在状态和统计。你可以传递一个列表的回调函数（作为 `callbacks` 关键字参数）到 `Sequential` 或 `Model` 类型的 `.fit()` 方法。在训练时，相应的回调函数的方法就会被在各自的阶段被调用。

**Callback**

```
keras.callbacks.callbacks.Callback()
```

用来组建新的回调函数的抽象基类。

## 属性

- **params**: 字典。训练参数，(例如，`verbosity`, `batch size`, `number of epochs...`)。
- **model**: `keras.models.Model` 的实例。指代被训练模型。

被回调函数作为参数的 `logs` 字典，它会含有于当前批量或训练轮相关数据的键。

目前，`Sequential` 模型类的 `.fit()` 方法会在传入到回调函数的 `logs` 里面包含以下的数据：

- **on\_epoch\_end**: 包括 `acc` 和 `loss` 的日志，也可以选择性的包括 `val_loss` (如果在 `fit` 中启用验证) , 和 `val_acc` (如果启用验证和监测精确值) 。
- **on\_batch\_begin**: 包括 `size` 的日志，在当前批量内的样本数量。
- **on\_batch\_end**: 包括 `loss` 的日志，也可以选择性的包括 `acc` (如果启用监测精确值) 。

BaseLogger

```
keras.callbacks.callbacks.BaseLogger(stateful_metrics=None)
```

会积累训练轮平均评估的回调函数。

这个回调函数被自动应用到每一个 Keras 模型上面。

## 参数

**stateful\_metrics**: 可重复使用不应在一个 epoch 上平均的指标的字符串名称。此列表中的度量标准将按原样记录在 `on_epoch_end` 中。所有其他指标将在 `on_epoch_end` 中取平均值。

**TerminateOnNaN**

```
keras.callbacks.callbacks.TerminateOnNaN()
```

当遇到 NaN 损失会停止训练的回调函数。

ProgbarLogger

```
keras.callbacks.callbacks.ProgbarLogger(count_mode='samples',  
stateful_metrics=None)
```

会把评估以标准输出打印的回调函数。

## 参数

- **count\_mode**: “steps” 或者 “samples”。进度条是否应该计数看见的样本或步骤（批量）。
- **stateful\_metrics**: 可重复使用不应在一个 epoch 上平均的指标的字符串名称。此列表中的度量标准将按原样记录在 `on_epoch_end` 中。所有其他指标将在 `on_epoch_end` 中取平均值。

## 异常

- **ValueError**: 如果 `count_mode`

History

```
keras.callbacks.callbacks.History()
```

把所有事件都记录到 `History` 对象的回调函数。

这个回调函数被自动启用到每一个 Keras 模型。`History` 对象会被模型的 `fit` 方法返回。

ModelCheckpoint

```
keras.callbacks.callbacks.ModelCheckpoint(filepath, monitor='val_loss',
verbose=0, save_best_only=False, save_weights_only=False, mode='auto',
period=1)
```

在每个训练期之后保存模型。

`filepath` 可以包括命名格式选项，可以由 `epoch` 的值和 `logs` 的键（由 `on_epoch_end` 参数传递）来填充。

例如：如果 `filepath` 是 `weights.{epoch:02d}-{val_loss:.2f}.hdf5`，那么模型被保存的文件名就会有训练轮数和验证损失。

## 参数

- **filepath**: 字符串，保存模型的路径。
- **monitor**: 被监测的数据。
- **verbose**: 详细信息模式，0 或者1。
- **save\_best\_only**: 如果 `save_best_only=True`，被监测数据的最佳模型就不会被覆盖。
- **mode**: {auto, min, max} 的其中之一。如果 `save_best_only=True`，那么是否覆盖保存文件的决定就取决于被监测数据的最大或者最小值。对于 `val_acc`，模式就会是 `max`，而对于 `val_loss`，模式就需要是 `min`，等等。在 `auto` 模式中，方向会自动从被监测的数据的名字中判断出来。
- **save\_weights\_only**: 如果 `True`，那么只有模型的权重会被保存(`model.save_weights(filepath)`)，否则的话，整个模型会被保存(`model.save(filepath)`)。
- **period**: 每个检查点之间的间隔(训练轮数)。

EarlyStopping

```
keras.callbacks.callbacks.EarlyStopping(monitor='val_loss', min_delta=0,  
patience=0, verbose=0, mode='auto', baseline=None, restore_best_weights=False)
```

当被监测的数量不再提升，则停止训练。

## 参数

- **monitor**: 被监测的数据。
- **min\_delta**: 在被监测的数据中被认为是提升的最小变化，例如，小于 min\_delta 的绝对变化会被认为没有提升。
- **patience**: 在监测质量经过多少轮次没有进度时即停止。如果验证频率 (`model.fit(validation_freq=5)`) 大于 1，则可能不会在每个轮次都产生验证数。
- **verbose**: 详细信息模式。
- **mode**: {auto, min, max} 其中之一。在 `min` 模式中，当被监测的数据停止下降，训练就会停止；在 `max` 模式中，当被监测的数据停止上升，训练就会停止；在 `auto` 模式中，方向会自动从被监测的数据的名字中判断出来。
- **baseline**: 要监控的数量的基准值。如果模型没有显示基准的改善，训练将停止。
- **restore\_best\_weights**: 是否从具有监测数量的最佳值的时期恢复模型权重。如果为 False，则使用在训练的最后一步获得的模型权重。

RemoteMonitor

```
keras.callbacks.callbacks.RemoteMonitor(root='http://localhost:9000', path='/  
publish/epoch/end/', field='data', headers=None, send_as_json=False)
```

将事件数据流到服务器的回调函数。

需要 `requests` 库。事件被默认发送到 `root + '/publish/epoch/end/'`。采用 HTTP POST，其中的 `data` 参数是以 JSON 编码的事件数据字典。如果 `send_as_json` 设置为 True，请求的 content type 是 `application/json`。否则，将在表单中发送序列化的 JSON。

## 参数

- **root**: 字符串；目标服务器的根地址。
- **path**: 字符串；相对于 `root` 的路径，事件数据被送达的地址。
- **field**: 字符串；JSON，数据被保存的领域。
- **headers**: 字典；可选自定义的 HTTP 的头字段。
- **send\_as\_json**: 布尔值；请求是否应该以 `application/json` 格式发送。

LearningRateScheduler

```
keras.callbacks.callbacks.LearningRateScheduler(schedule, verbose=0)
```

学习速率定时器。

## 参数

- **schedule**: 一个函数，接受轮索引数作为输入（整数，从 0 开始迭代）然后返回一个学习速率作为输出（浮点数）。
- **verbose**: 整数。0：安静，1：更新信息。

ReduceLROnPlateau

```
keras.callbacks.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.1,  
patience=10, verbose=0, mode='auto', min_delta=0.0001, cooldown=0, min_lr=0)
```

当标准评估停止提升时，降低学习速率。

当学习停止时，模型总是会受益于降低 2-10 倍的学习速率。这个回调函数监测一个数据并且当这个数据在一定「有耐心」的训练轮之后还没有进步，那么学习速率就会被降低。

## 例子

```
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                               patience=5, min_lr=0.001)
model.fit(X_train, Y_train, callbacks=[reduce_lr])
```

## 参数

- **monitor**: 被监测的数据。
- **factor**: 学习速率被降低的因数。新的学习速率 = 学习速率 \* 因数
- **patience**: 在监测质量经过多少轮次没有进度时即停止。如果验证频率  
(`model.fit(validation_freq=5)`) 大于 1，则可能不会在每个轮次都产生验证数。
- **verbose**: 整数。0: 安静, 1: 更新信息。
- **mode**: {auto, min, max} 其中之一。如果是 `min` 模式，学习速率会被降低如果被监测的数据已经停止下降；在 `max` 模式，学习塑料会被降低如果被监测的数据已经停止上升；在 `auto` 模式，方向会被从被监测的数据中自动推断出来。
- **min\_delta**: 对于测量新的最优化的阀值，只关注巨大的改变。
- **cooldown**: 在学习速率被降低之后，重新恢复正常操作之前等待的训练轮数量。
- **min\_lr**: 学习速率的下边界。

CSVLogger

```
keras.callbacks.callbacks.CSVLogger(filename, separator=',', append=False)
```

把训练轮结果数据流到 csv 文件的回调函数。

支持所有可以被作为字符串表示的值，包括 1D 可迭代数据，例如，`np.ndarray`。

## 例子

```
csv_logger = CSVLogger('training.log')
model.fit(X_train, Y_train, callbacks=[csv_logger])
```

## 参数

- **filename**: csv 文件的文件名，例如 'run/log.csv'。
- **separator**: 用来隔离 csv 文件中元素的字符串。
- **append**: True: 如果文件存在则增加（可以被用于继续训练）。False: 覆盖存在的文件。

LambdaCallback

```
keras.callbacks.callbacks.LambdaCallback(on_epoch_begin=None,  
on_epoch_end=None, on_batch_begin=None, on_batch_end=None,  
on_train_begin=None, on_train_end=None)
```

在训练进行中创建简单，自定义的回调函数的回调函数。

这个回调函数和匿名函数在合适的时间被创建。需要注意的是回调函数要求位置型参数，如下：

- `on_epoch_begin` 和 `on_epoch_end` 要求两个位置型的参数：`epoch`, `logs`
- `on_batch_begin` 和 `on_batch_end` 要求两个位置型的参数：`batch`, `logs`
- `on_train_begin` 和 `on_train_end` 要求一个位置型的参数：`logs`

## 参数

- `on_epoch_begin`: 在每轮开始时被调用。
- `on_epoch_end`: 在每轮结束时被调用。
- `on_batch_begin`: 在每批开始时被调用。
- `on_batch_end`: 在每批结束时被调用。
- `on_train_begin`: 在模型训练开始时被调用。
- `on_train_end`: 在模型训练结束时被调用。

## 示例

```
# 在每一个批开始时，打印出批数。
batch_print_callback = LambdaCallback(
    on_batch_begin=lambda batch, logs: print(batch))

# 把训练轮损失数据流到 JSON 格式的文件。文件的内容
# 不是完美的 JSON 格式，但是每行都是 JSON 对象。
import json
json_log = open('loss_log.json', mode='wt', buffering=1)
json_logging_callback = LambdaCallback(
    on_epoch_end=lambda epoch, logs: json_log.write(
        json.dumps({'epoch': epoch, 'loss': logs['loss']})) + '\n'),
    on_train_end=lambda logs: json_log.close()
)

# 在完成模型训练之后，结束一些进程。
processes = ...
cleanup_callback = LambdaCallback(
    on_train_end=lambda logs: [
        p.terminate() for p in processes if p.is_alive()])

model.fit(...,
          callbacks=[batch_print_callback,
                     json_logging_callback,
                     cleanup_callback])
```

## TensorBoard

[\[source\]](#)

```
keras.callbacks.tensorboard_v1.TensorBoard(log_dir='./logs', histogram_freq=0,
batch_size=32, write_graph=True, write_grads=False, write_images=False,
embeddings_freq=0, embeddings_layer_names=None, embeddings_metadata=None,
embeddings_data=None, update_freq='epoch')
```

Tensorboard 基本可视化。

[TensorBoard](#) 是由 Tensorflow 提供的一个可视化工具。

这个回调函数为 Tensorboard 编写一个日志，这样你可以可视化测试和训练的标准评估的动态图像，也可以可视化模型中不同层的激活值直方图。

如果你已经使用 pip 安装了 Tensorflow，你应该可以从命令行启动 Tensorflow：

```
tensorboard --logdir=/full_path_to_your_logs
```

当使用 TensorFlow 之外的后端时，TensorBoard 仍然可以运行（如果你安装了 TensorFlow），但是仅有展示损失值和评估指标这两个功能可用。

## 参数

- **log\_dir**: 用来保存被 TensorBoard 分析的日志文件的文件名。
- **histogram\_freq**: 对于模型中各个层计算激活值和模型权重直方图的频率（训练轮数中）。如果设置成 0，直方图不会被计算。对于直方图可视化的验证数据（或分离数据）一定要明确的指出。
- **batch\_size**: 用以直方图计算的传入神经元网络输入批的大小。
- **write\_graph**: 是否在 TensorBoard 中可视化图像。如果 `write_graph` 被设置为 `True`，日志文件会变得非常大。
- **write\_grads**: 是否在 TensorBoard 中可视化梯度值直方图。`histogram_freq` 必须要大于 0。
- **write\_images**: 是否在 TensorBoard 中将模型权重以图片可视化。
- **embeddings\_freq**: 被选中的嵌入层会被保存的频率（在训练轮中）。如果设置为 0，则不会计算嵌入。要在 TensorBoard 的嵌入选项卡中可视化的数据必须作为 `embeddings_data` 传递。
- **embeddings\_layer\_names**: 一个列表，会被监测层的名字。如果是 `None` 或空列表，那么所有的嵌入层都会被监测。
- **embeddings\_metadata**: 一个字典，对应层的名字到保存有这个嵌入层元数据文件的名字。查看[详情](#) 关于元数据的数据格式。以防同样的元数据被用于所用的嵌入层，字符串可以被传入。
- **embeddings\_data**: 要嵌入在 `embeddings_layer_names` 指定的层的数据。Numpy 数组（如果模型有单个输入）或 Numpy 数组列表（如果模型有多个输入）。[Learn more about embeddings](#)。
- **update\_freq**: '`batch`' 或 '`epoch`' 或整数。当使用 '`batch`' 时，在每个 batch 之后将损失和评估值写入到 TensorBoard 中。同样的情况应用到 '`epoch`' 中。如果使用整数，例如 `10000`，这个回调会在每 10000 个样本之后将损失和评估值写入到 TensorBoard 中。注意，频繁地写入到 TensorBoard 会减缓你的训练。

## 创建一个回调

你可以通过扩展 `keras.callbacks.Callback` 基类来创建一个自定义的回调函数。通过类的属性 `self.model`，回调函数可以获得它所联系的模型。

下面是一个简单的例子，在训练时，保存一个列表的批量损失值：

```

class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))

```

## 示例: 记录损失历史

```

class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))

model = Sequential()
model.add(Dense(10, input_dim=784, kernel_initializer='uniform'))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

history = LossHistory()
model.fit(x_train, y_train, batch_size=128, epochs=20, verbose=0,
callbacks=[history])

print(history.losses)
# 输出
...
[0.66047596406559383, 0.3547245744908703, ..., 0.25953155204159617,
0.25901699725311789]
...

```

## 示例: 模型检查点

```

from keras.callbacks import ModelCheckpoint

model = Sequential()
model.add(Dense(10, input_dim=784, kernel_initializer='uniform'))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

...
如果验证损失下降， 那么在每个训练轮之后保存模型。
...

checkpointer = ModelCheckpoint(filepath='/tmp/weights.hdf5', verbose=1,
save_best_only=True)
model.fit(x_train, y_train, batch_size=128, epochs=20, verbose=0,
validation_data=(X_test, Y_test), callbacks=[checkpointer])

```



# 数据集

## CIFAR10 小图像分类数据集

50,000 张 32x32 彩色训练图像数据，以及 10,000 张测试图像数据，总共分为 10 个类别。

用法：

```
from keras.datasets import cifar10  
  
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

- **返回：**
- 2 个元组：
  - **x\_train, x\_test:** uint8 数组表示的 RGB 图像数据，尺寸为 (num\_samples, 3, 32, 32) 或 (num\_samples, 32, 32, 3)，基于 `image_data_format` 后端设定的 `channels_first` 或 `channels_last`。
  - **y\_train, y\_test:** uint8 数组表示的类别标签（范围在 0-9 之间的整数），尺寸为 (num\_samples, 1)。

## CIFAR100 小图像分类数据集

50,000 张 32x32 彩色训练图像数据，以及 10,000 张测试图像数据，总共分为 100 个类别。

用法：

```
from keras.datasets import cifar100  
  
(x_train, y_train), (x_test, y_test) = cifar100.load_data(label_mode='fine')
```

- **返回：**
- 2 个元组：
  - **x\_train, x\_test:** uint8 数组表示的 RGB 图像数据，尺寸为 (num\_samples, 3, 32, 32) 或 (num\_samples, 32, 32, 3)，基于 `image_data_format` 后端设定的 `channels_first` 或 `channels_last`。
  - **y\_train, y\_test:** uint8 数组表示的类别标签，尺寸为 (num\_samples, 1)。

• 参数:

- **label\_mode**: “fine” 或者 “coarse”

## IMDB 电影评论情感分类数据集

数据集来自 IMDB 的 25,000 条电影评论，以情绪（正面/负面）标记。评论已经过预处理，并编码为词索引（整数）的序列表示。为了方便起见，将词按数据集中出现的频率进行索引，例如整数 3 编码数据中第三个最频繁的词。这允许快速筛选操作，例如：「只考虑前 10,000 个最常用的词，但排除前 20 个最常见的词」。

作为惯例，0 不代表特定的单词，而是被用于编码任何未知单词。

## 用法

```
from keras.datasets import imdb

(x_train, y_train), (x_test, y_test) = imdb.load_data(path="imdb.npz",
                                                       num_words=None,
                                                       skip_top=0,
                                                       maxlen=None,
                                                       seed=113,
                                                       start_char=1,
                                                       oov_char=2,
                                                       index_from=3)
```

• 返回:

- 2 个元组:

- **x\_train, x\_test**: 序列的列表，即词索引的列表。如果指定了 `num_words` 参数，则可能的最大索引值是 `num_words-1`。如果指定了 `maxlen` 参数，则可能的最大序列长度为 `maxlen`。

- **y\_train, y\_test**: 整数标签列表 (1 或 0)。

• 参数:

- **path**: 如果你本地没有该数据集 (在 `'~/keras/datasets/' + path`)，它将被下载到此目录。
- **num\_words**: 整数或 None。要考虑的最常用的词语。任何不太频繁的词将在序列数据中显示为 `oov_char` 值。
- **skip\_top**: 整数。要忽略的最常见的单词（它们将在序列数据中显示为 `oov_char` 值）。
- **maxlen**: 整数。最大序列长度。任何更长的序列都将被截断。
- **seed**: 整数。用于可重现数据混洗的种子。

- **start\_char**: 整数。序列的开始将用这个字符标记。设置为 1，因为 0 通常作为填充字符。
- **oov\_char**: 整数。由于 `num_words` 或 `skip_top` 限制而被删除的单词将被替换为此字符。
- **index\_from**: 整数。使用此数以上更高的索引值实际词汇索引的开始。

## 路透社新闻主题分类

数据集来源于路透社的 11,228 条新闻文本，总共分为 46 个主题。与 IMDB 数据集一样，每条新闻都被编码为一个词索引的序列（相同的约定）。

用法：

```
from keras.datasets import reuters

(x_train, y_train), (x_test, y_test) = reuters.load_data(path="reuters.npz",
                                                       num_words=None,
                                                       skip_top=0,
                                                       maxlen=None,
                                                       test_split=0.2,
                                                       seed=113,
                                                       start_char=1,
                                                       oov_char=2,
                                                       index_from=3)
```

规格与 IMDB 数据集的规格相同，但增加了：

- **test\_split**: 浮点型。用作测试集的数据比例。

该数据集还提供了用于编码序列的词索引：

```
word_index = reuters.get_word_index(path="reuters_word_index.json")
```

- **返回**：一个字典，其中键是单词（字符串），值是索引（整数）。例如，`word_index["giraffe"]` 可能会返回 1234。

### • 参数：

- **path**: 如果在本地没有索引文件 (at `'~/keras/datasets/' + path`)，它将被下载到该目录。

## MNIST 手写字符数据集

训练集为 60,000 张 28x28 像素灰度图像，测试集为 10,000 同规格图像，总共 10 类数字标签。

用法：

```
from keras.datasets import mnist  
  
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

· 返回：

· 2 个元组：

- **x\_train, x\_test**: uint8 数组表示的灰度图像，尺寸为 (num\_samples, 28, 28)。
- **y\_train, y\_test**: uint8 数组表示的数字标签（范围在 0-9 之间的整数），尺寸为 (num\_samples,)。

· 参数：

- **path**: 如果在本地没有索引文件 (at `'~/keras/datasets/' + path`)，它将被下载到该目录。

## Fashion-MNIST 时尚物品数据集

训练集为 60,000 张 28x28 像素灰度图像，测试集为 10,000 同规格图像，总共 10 类时尚物品标签。该数据集可以用作 MNIST 的直接替代品。类别标签是：

类别	描述	中文
0	T-shirt/top	T恤/上衣
1	Trouser	裤子
2	Pullover	套头衫
3	Dress	连衣裙
4	Coat	外套
5	Sandal	凉鞋

类别	描述	中文
6	Shirt	衬衫
7	Sneaker	运动鞋
8	Bag	背包
9	Ankle boot	短靴

用法：

```
from keras.datasets import fashion_mnist
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
```

· 返回：

· 2 个元组：

- **x\_train, x\_test**: uint8 数组表示的灰度图像，尺寸为 (num\_samples, 28, 28)。
- **y\_train, y\_test**: uint8 数组表示的数字标签（范围在 0-9 之间的整数），尺寸为 (num\_samples,)。

## Boston 房价回归数据集

数据集来自卡内基梅隆大学维护的 StatLib 库。

样本包含 1970 年代的在波士顿郊区不同位置的房屋信息，总共有 13 种房屋属性。目标值是一个位置的房屋的中值（单位：k\$）。

用法：

```
from keras.datasets import boston_housing
(x_train, y_train), (x_test, y_test) = boston_housing.load_data()
```

· 参数：

- **path**: 缓存本地数据集的位置 (相对路径 ~/.keras/datasets)。
- **seed**: 在计算测试分割之前对数据进行混洗的随机种子。

- **test\_split**: 需要保留作为测试数据的比例。
- **返回**: Numpy 数组的元组: `(x_train, y_train), (x_test, y_test)`。



# 应用 Applications

Keras 的应用模块 (`keras.applications`) 提供了带有预训练权值的深度学习模型，这些模型可以用来进行预测、特征提取和微调（fine-tuning）。

当你初始化一个预训练模型时，会自动下载权重到 `~/.keras/models/` 目录下。

## 可用的模型

在 ImageNet 上预训练过的用于图像分类的模型：

- [Xception](#)
- [VGG16](#)
- [VGG19](#)
- [ResNet, ResNetV2](#)
- [InceptionV3](#)
- [InceptionResNetV2](#)
- [MobileNet](#)
- [MobileNetV2](#)
- [DenseNet](#)
- [NASNet](#)

所有的这些架构都兼容所有的后端 (TensorFlow, Theano 和 CNTK)，并且会在实例化时，根据 Keras 配置文件 `~/.keras/keras.json` 中设置的图像数据格式构建模型。举个例子，如果你设置 `image_data_format=channels_last`，则加载的模型将按照 TensorFlow 的维度顺序来构造，即「高度-宽度-深度」(Height-Width-Depth) 的顺序。

注意：

- 对于 `Keras < 2.2.0`，`Xception` 模型仅适用于 TensorFlow，因为它依赖于 `SeparableConvolution` 层。
- 对于 `Keras < 2.1.5`，`MobileNet` 模型仅适用于 TensorFlow，因为它依赖于 `DepthwiseConvolution` 层。

## 图像分类模型的使用示例

### 使用 ResNet50 进行 ImageNet 分类

```
from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input, decode_predictions
import numpy as np

model = ResNet50(weights='imagenet')

img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

preds = model.predict(x)
# 将结果解码为元组列表 (class, description, probability)
# (一个列表代表批次中的一个样本)
print('Predicted:', decode_predictions(preds, top=3)[0])
# Predicted: [(u'n02504013', u'Indian_elephant', 0.82658225), (u'n01871265',
u'tusker', 0.1122357), (u'n02504458', u'African_elephant', 0.061040461)]
```

### 使用 VGG16 提取特征

```
from keras.applications.vgg16 import VGG16
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
import numpy as np

model = VGG16(weights='imagenet', include_top=False)

img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

features = model.predict(x)
```

### 从VGG19 的任意中间层中抽取特征

```
from keras.applications.vgg19 import VGG19
from keras.preprocessing import image
from keras.applications.vgg19 import preprocess_input
from keras.models import Model
import numpy as np

base_model = VGG19(weights='imagenet')
model = Model(inputs=base_model.input,
```

```

outputs=base_model.get_layer('block4_pool').output)

img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

block4_pool_features = model.predict(x)

```

## 在新类上微调 InceptionV3

```

from keras.applications.inception_v3 import InceptionV3
from keras.preprocessing import image
from keras.models import Model
from keras.layers import Dense, GlobalAveragePooling2D
from keras import backend as K

# 构建不带分类器的预训练模型
base_model = InceptionV3(weights='imagenet', include_top=False)

# 添加全局平均池化层
x = base_model.output
x = GlobalAveragePooling2D()(x)

# 添加一个全连接层
x = Dense(1024, activation='relu')(x)

# 添加一个分类器，假设我们有200个类
predictions = Dense(200, activation='softmax')(x)

# 构建我们需要训练的完整模型
model = Model(inputs=base_model.input, outputs=predictions)

# 首先，我们只训练顶部的几层（随机初始化的层）
# 锁住所有 InceptionV3 的卷积层
for layer in base_model.layers:
    layer.trainable = False

# 编译模型（一定要在锁层以后操作）
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

# 在新的数据集上训练几代
model.fit_generator(...)

# 现在顶层应该训练好了，让我们开始微调 Inception V3 的卷积层。
# 我们会锁住底下的几层，然后训练其余的顶层。

# 让我们看看每一层的名字和层号，看看我们应该锁多少层呢：
for i, layer in enumerate(base_model.layers):
    print(i, layer.name)

# 我们选择训练最上面的两个 Inception block
# 也就是说锁住前面249层，然后放开之后的层。
for layer in model.layers[:249]:

```

```

        layer.trainable = False
for layer in model.layers[249:]:
    layer.trainable = True

# 我们需要重新编译模型，才能使上面的修改生效
# 让我们设置一个很低的学习率，使用 SGD 来微调
from keras.optimizers import SGD
model.compile(optimizer=SGD(lr=0.0001, momentum=0.9),
loss='categorical_crossentropy')

# 我们继续训练模型，这次我们训练最后两个 Inception block
# 和两个全连接层
model.fit_generator(...)
```

## 通过自定义输入张量构建 InceptionV3

```

from keras.applications.inception_v3 import InceptionV3
from keras.layers import Input

# 这也可能是不同的 Keras 模型或层的输出
input_tensor = Input(shape=(224, 224, 3)) # 假定 K.image_data_format() ==
'channels_last'

model = InceptionV3(input_tensor=input_tensor, weights='imagenet',
include_top=True)
```

## 模型概览

模型	大小	Top-1 准确率	Top-5 准确率	参数数量	深度
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-

模型	大小	Top-1 准确率	Top-5 准确率	参数数量	深度
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-

Top-1 准确率和 Top-5 准确率都是在 ImageNet 验证集上的结果。

Depth 表示网络的拓扑深度。这包括激活层，批标准化层等。

Xception

```
keras.applications.xception.Xception(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)
```

在 ImageNet 上预训练的 Xception V1 模型。

在 ImageNet 上，该模型取得了验证集 top1 0.790 和 top5 0.945 的准确率。

该模型可同时构建于 `channels_first` (通道, 高度, 宽度) 和 `channels_last` (高度, 宽度, 通道) 两种输入维度顺序。

模型默认输入尺寸是 299x299。

## 参数

- **include\_top**: 是否包括顶层的全连接层。
- **weights**: `None` 代表随机初始化，`'imagenet'` 代表加载在 ImageNet 上预训练的权值。
- **input\_tensor**: 可选，Keras tensor 作为模型的输入（即 `layers.Input()` 输出的 tensor）。
- **input\_shape**: 可选，输入尺寸元组，仅当 `include_top=False` 时有效（否则输入形状必须是 `(299, 299, 3)`，因为预训练模型是以这个大小训练的）。它必须拥有 3 个输入通道，且宽高必须不小于 71。例如 `(150, 150, 3)` 是一个合法的输入尺寸。
- **pooling**: 可选，当 `include_top` 为 `False` 时，该参数指定了特征提取时的池化方式。
  - `None` 代表不池化，直接输出最后一层卷积块的输出，该输出是一个 4D 张量。
  - `'avg'` 代表全局平均池化 (GlobalAveragePooling2D)，相当于在最后一层卷积块后面再加一层全局平均池化层，输出是一个 2D 张量。
  - `'max'` 代表全局最大池化。
- **classes**: 可选，图片分类的类别数，仅当 `include_top` 为 `True` 并且不加载预训练权值时可用。

## 返回

一个 Keras Model 对象。

## 参考文献

- [Xception: Deep Learning with Depthwise Separable Convolutions](#)

## License

预训练权值由我们自己训练而来，基于 MIT license 发布。

# VGG16

```
keras.applications.vgg16.VGG16(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)
```

VGG16 模型，权值由 ImageNet 训练而来。

该模型可同时构建于 `channels_first` (通道, 高度, 宽度) 和 `channels_last` (高度, 宽度, 通道) 两种输入维度顺序。

模型默认输入尺寸是 224x224。

## 参数

- **include\_top**: 是否包括顶层的全连接层。
- **weights**: `None` 代表随机初始化, `'imagenet'` 代表加载在 ImageNet 上预训练的权值。
- **input\_tensor**: 可选, Keras tensor 作为模型的输入 (即 `layers.Input()` 输出的 tensor)。
- **input\_shape**: 可选, 输入尺寸元组, 仅当 `include_top=False` 时有效, 否则输入形状必须是 `(244, 244, 3)` (对于 `channels_last` 数据格式), 或者 `(3, 244, 244)` (对于 `channels_first` 数据格式)。它必须拥有 3 个输入通道, 且宽高必须不小于 32。例如 `(200, 200, 3)` 是一个合法的输入尺寸。
- **pooling**: 可选, 当 `include_top` 为 `False` 时, 该参数指定了特征提取时的池化方式。
  - `None` 代表不池化, 直接输出最后一层卷积块的输出, 该输出是一个 4D 张量。
  - `'avg'` 代表全局平均池化 (GlobalAveragePooling2D), 相当于在最后一层卷积块后面再加一层全局平均池化层, 输出是一个二维张量。
  - `'max'` 代表全局最大池化。
- **classes**: 可选, 图片分类的类别数, 仅当 `include_top` 为 `True` 并且不加载预训练权值时可用。

## 返回

一个 Keras Model 对象。

## 参考文献

- [Very Deep Convolutional Networks for Large-Scale Image Recognition](#): 如果在研究中使用了VGG, 请引用该论文。

## License

预训练权值由 [VGG at Oxford](#) 发布的预训练权值移植而来, 基于 [Creative Commons Attribution License](#)。

# VGG19

```
keras.applications.vgg19.VGG19(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)
```

VGG19 模型，权值由 ImageNet 训练而来。

该模型可同时构建于 `channels_first` (通道, 高度, 宽度) 和 `channels_last` (高度, 宽度, 通道) 两种输入维度顺序。

模型默认输入尺寸是 224x224。

## 参数

- **include\_top**: 是否包括顶层的全连接层。
- **weights**: `None` 代表随机初始化，`'imagenet'` 代表加载在 ImageNet 上预训练的权值。
- **input\_tensor**: 可选，Keras tensor 作为模型的输入（即 `layers.Input()` 输出的 tensor）。
- **input\_shape**: 可选，输入尺寸元组，仅当 `include_top=False` 时有效，否则输入形状必须是 `(224, 224, 3)` (对于 `channels_last` 数据格式)，或者 `(3, 224, 224)` (对于 `channels_first` 数据格式)。它必须拥有 3 个输入通道，且宽高必须不小于 32。例如 `(200, 200, 3)` 是一个合法的输入尺寸。
- **pooling**: 可选，当 `include_top` 为 `False` 时，该参数指定了特征提取时的池化方式。
  - `None` 代表不池化，直接输出最后一层卷积层的输出，该输出是一个四维张量。
  - `'avg'` 代表全局平均池化 (GlobalAveragePooling2D)，相当于在最后一层卷积层后面再加一层全局平均池化层，输出是一个二维张量。
  - `'max'` 代表全局最大池化
- **classes**: 可选，图片分类的类别数，仅当 `include_top` 为 `True` 并且不加载预训练权值时可用。

## 返回

一个 Keras Model 对象。

## 参考文献

- [Very Deep Convolutional Networks for Large-Scale Image Recognition](#): 如果在研究中使用了VGG，请引用该论文。

## License

预训练权值由 [VGG at Oxford](#) 发布的预训练权值移植而来，基于 [Creative Commons Attribution License](#)。

## ResNet

```
keras.applications.resnet.ResNet50(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)
keras.applications.resnet.ResNet101(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)
keras.applications.resnet.ResNet152(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)
keras.applications.resnet_v2.ResNet50V2(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)
keras.applications.resnet_v2.ResNet101V2(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)
keras.applications.resnet_v2.ResNet152V2(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)
```

ResNet, ResNetV2 模型，权值由 ImageNet 训练而来。

该模型可同时构建于 `channels_first` (通道, 高度, 宽度) 和 `channels_last` (高度, 宽度, 通道) 两种输入维度顺序。

模型默认输入尺寸是 224x224。

### 参数

- **include\_top**: 是否包括顶层的全连接层。
- **weights**: `None` 代表随机初始化，`'imagenet'` 代表加载在 ImageNet 上预训练的权值。
- **input\_tensor**: 可选，Keras tensor 作为模型的输入（即 `layers.Input()` 输出的 tensor）。
- **input\_shape**: 可选，输入尺寸元组，仅当 `include_top=False` 时有效，否则输入形状必须是 `(244, 244, 3)` (对于 `channels_last` 数据格式)，或者 `(3, 244, 244)` (对于 `channels_first` 数据格式)。它必须拥有 3 个输入通道，且宽高必须不小于 32。例如 `(200, 200, 3)` 是一个合法的输入尺寸。
- **pooling**: 可选，当 `include_top` 为 `False` 时，该参数指定了特征提取时的池化方式。
  - `None` 代表不池化，直接输出最后一层卷积块的输出，该输出是一个 4D 张量。
  - `'avg'` 代表全局平均池化 (GlobalAveragePooling2D)，相当于在最后一层卷积块后面再加一层全局平均池化层，输出是一个二维张量。
  - `'max'` 代表全局最大池化
- **classes**: 可选，图片分类的类别数，仅当 `include_top` 为 `True` 并且不加载预训练权值时可用。

## 返回

一个 Keras Model 对象。

## 参考文献

- ResNet : [Deep Residual Learning for Image Recognition](#)
- ResNetV2 : [Identity Mappings in Deep Residual Networks](#)

## License

预训练权值由以下提供：

- ResNet : [The original repository of Kaiming He](#) under the [MIT license](#).
- ResNetV2 : [Facebook](#) under the [BSD license](#).

## InceptionV3

```
keras.applications.inception_v3.InceptionV3(include_top=True,  
weights='imagenet', input_tensor=None, input_shape=None, pooling=None,  
classes=1000)
```

Inception V3 模型，权值由 ImageNet 训练而来。

该模型可同时构建于 `channels_first` (通道, 高度, 宽度) 和 `channels_last` (高度, 宽度, 通道) 两种输入维度顺序。

模型默认输入尺寸是 299x299。

## 参数

- **include\_top**: 是否包括顶层的全连接层。
- **weights**: `None` 代表随机初始化，`'imagenet'` 代表加载在 ImageNet 上预训练的权值。
- **input\_tensor**: 可选，Keras tensor 作为模型的输入（即 `layers.Input()` 输出的 tensor）。
- **input\_shape**: 可选，输入尺寸元组，仅当 `include_top=False` 时有效，否则输入形状必须是 `(299, 299, 3)` (对于 `channels_last` 数据格式)，或者 `(3, 299, 299)` (对于 `channels_first` 数据格式)。它必须拥有 3 个输入通道，且宽高必须不小于 139。例如 `(150, 150, 3)` 是一个合法的输入尺寸。
- **pooling**: 可选，当 `include_top` 为 `False` 时，该参数指定了特征提取时的池化方式。
  - `None` 代表不池化，直接输出最后一层卷积块的输出，该输出是一个 4D 张量。

- `'avg'` 代表全局平均池化 (GlobalAveragePooling2D)，相当于在最后一层卷积块后面再加一层全局平均池化层，输出是一个二维张量。
  - `'max'` 代表全局最大池化。
- **classes**: 可选，图片分类的类别数，仅当 `include_top` 为 `True` 并且不加载预训练权值时可用。

## 返回

一个 Keras `Model` 对象。

## 参考文献

- [Rethinking the Inception Architecture for Computer Vision](#)

## License

预训练权值基于 [Apache License](#)。

## InceptionResNetV2

```
keras.applications.inception_resnet_v2.InceptionResNetV2(include_top=True,  
weights='imagenet', input_tensor=None, input_shape=None, pooling=None,  
classes=1000)
```

Inception-ResNet V2 模型，权值由 ImageNet 训练而来。

该模型可同时构建于 `channels_first` (通道, 高度, 宽度) 和 `channels_last` (高度, 宽度, 通道) 两种输入维度顺序。

模型默认输入尺寸是 299x299。

## 参数

- **include\_top**: 是否包括顶层的全连接层。
- **weights**: `None` 代表随机初始化，`'imagenet'` 代表加载在 ImageNet 上预训练的权值。
- **input\_tensor**: 可选，Keras tensor 作为模型的输入（即 `layers.Input()` 输出的 tensor）。
- **input\_shape**: 可选，输入尺寸元组，仅当 `include_top=False` 时有效，否则输入形状必须是 `(299, 299, 3)` (对于 `channels_last` 数据格式)，或者 `(3, 299, 299)` (对于 `channels_first` 数据格式)。它必须拥有 3 个输入通道，且宽高必须不小于 139。例如 `(150, 150, 3)` 是一个合法的输入尺寸。
- **pooling**: 可选，当 `include_top` 为 `False` 时，该参数指定了特征提取时的池化方式。
  - `None` 代表不池化，直接输出最后一层卷积块的输出，该输出是一个 4D 张量。

- `'avg'` 代表全局平均池化 (GlobalAveragePooling2D)，相当于在最后一层卷积块后面再加一层全局平均池化层，输出是一个二维张量。
  - `'max'` 代表全局最大池化
- **classes**: 可选，图片分类的类别数，仅当 `include_top` 为 `True` 并且不加载预训练权值时可用。

## 返回

一个 Keras `Model` 对象。

## 参考文献

- [Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning](#)

## License

预训练权值基于 [Apache License](#)。

## MobileNet

```
keras.applications.mobilenet.MobileNet(input_shape=None, alpha=1.0,
depth_multiplier=1, dropout=1e-3, include_top=True, weights='imagenet',
input_tensor=None, pooling=None, classes=1000)
```

在 ImageNet 上预训练的 MobileNet 模型。

该模型可同时构建于 `channels_first` (通道, 高度, 宽度) 和 `channels_last` (高度, 宽度, 通道) 两种输入维度顺序。

模型默认输入尺寸是 224x224。

## 参数

- **input\_shape**: 可选，输入尺寸元组，仅当 `include_top=False` 时有效，否则输入形状必须是 `(224, 224, 3)` (`channels_last` 格式) 或 `(3, 224, 224)` (`channels_first` 格式)。它必须为 3 个输入通道，且宽高必须不小于 32，比如 `(200, 200, 3)` 是一个合法的输入尺寸。
- **alpha**: 控制网络的宽度：
  - 如果 `alpha < 1.0`，则同比例减少每层的滤波器个数。
  - 如果 `alpha > 1.0`，则同比例增加每层的滤波器个数。
  - 如果 `alpha = 1`，使用论文默认的滤波器个数
- **depth\_multiplier**: depthwise 卷积的深度乘子，也称为 (分辨率乘子)

- **dropout**: dropout 概率
- **include\_top**: 是否包括顶层的全连接层。
- **weights**: `None` 代表随机初始化，`'imagenet'` 代表加载在 ImageNet 上预训练的权值。
- **input\_tensor**: 可选，Keras tensor 作为模型的输入（比如 `layers.Input()` 输出的 tensor）。
- **pooling**: 可选，当 `include_top` 为 `False` 时，该参数指定了特征提取时的池化方式。
  - `None` 代表不池化，直接输出最后一层卷积块的输出，该输出是一个 4D 张量。
  - `'avg'` 代表全局平均池化（GlobalAveragePooling2D），相当于在最后一层卷积块后面再加一层全局平均池化层，输出是一个二维张量。
  - `'max'` 代表全局最大池化
- **classes**: 可选，图片分类的类别数，仅当 `include_top` 为 `True` 并且不加载预训练权值时可用。

## 返回

一个 Keras Model 对象。

## 参考文献

- [MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications](#)

## License

预训练权值基于 [Apache License](#)。

## DenseNet

```
keras.applications.densenet.DenseNet121(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)
keras.applications.densenet.DenseNet169(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)
keras.applications.densenet.DenseNet201(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)
```

在 ImageNet 上预训练的 DenseNet 模型。

该模型可同时构建于 `channels_first` (通道, 高度, 宽度) 和 `channels_last` (高度, 宽度, 通道) 两种输入维度顺序。

模型默认输入尺寸是 224x224。

## 参数

- **blocks**: 四个 Dense Layers 的 block 数量。
- **include\_top**: 是否包括顶层的全连接层。
- **weights**: `None` 代表随机初始化，`'imagenet'` 代表加载在 ImageNet 上预训练的权值。
- **input\_tensor**: 可选，Keras tensor 作为模型的输入（比如 `layers.Input()` 输出的 tensor）。
- **input\_shape**: 可选，输入尺寸元组，仅当 `include_top=False` 时有效（不然输入形状必须是 `(224, 224, 3)` (`channels_last` 格式) 或 `(3, 224, 224)` (`channels_first` 格式)，因为预训练模型是以这个大小训练的）。它必须为 3 个输入通道，且宽高必须不小于 32，比如 `(200, 200, 3)` 是一个合法的输入尺寸。
- **pooling**: 可选，当 `include_top` 为 `False` 时，该参数指定了特征提取时的池化方式。
  - `None` 代表不池化，直接输出最后一层卷积层的输出，该输出是一个四维张量。
  - `'avg'` 代表全局平均池化 (GlobalAveragePooling2D)，相当于在最后一层卷积层后面再加一层全局平均池化层，输出是一个二维张量。
  - `'max'` 代表全局最大池化。
- **classes**: 可选，图片分类的类别数，仅当 `include_top` 为 `True` 并且不加载预训练权值时可用。

## 返回

一个 Keras `Model` 对象。

## 参考文献

- [Densely Connected Convolutional Networks](#) (CVPR 2017 Best Paper Award)

## Licence

预训练权值基于 [BSD 3-clause License](#)。

## NASNet

```
keras.applications.nasnet.NASNetLarge(input_shape=None, include_top=True,
weights='imagenet', input_tensor=None, pooling=None, classes=1000)
keras.applications.nasnet.NASNetMobile(input_shape=None, include_top=True,
weights='imagenet', input_tensor=None, pooling=None, classes=1000)
```

在 ImageNet 上预训练的神经结构搜索网络模型 (NASNet)。

NASNetLarge 模型默认的输入尺寸是 331x331，NASNetMobile 模型默认的输入尺寸是 224x224。

## 参数

- **input\_shape**: 可选，输入尺寸元组，仅当 `include_top=False` 时有效，否则对于 NASNetMobile 模型来说，输入形状必须是 `(224, 224, 3)` (`channels_last` 格式) 或 `(3, 224, 224)` (`channels_first` 格式)，对于 NASNetLarge 来说，输入形状必须是 `(331, 331, 3)` (`channels_last` 格式) 或 `(3, 331, 331)` (`channels_first` 格式)。它必须为 3 个输入通道，且宽高必须不小于 32，比如 `(200, 200, 3)` 是一个合法的输入尺寸。
- **include\_top**: 是否包括顶层的全连接层。
- **weights**: `None` 代表随机初始化，`'imagenet'` 代表加载在 ImageNet 上预训练的权值。
- **input\_tensor**: 可选，Keras tensor 作为模型的输入（比如 `layers.Input()` 输出的 tensor）。
- **pooling**: 可选，当 `include_top` 为 `False` 时，该参数指定了特征提取时的池化方式。
  - `None` 代表不池化，直接输出最后一层卷积层的输出，该输出是一个四维张量。
  - `'avg'` 代表全局平均池化 (GlobalAveragePooling2D)，相当于在最后一层卷积层后面再加一层全局平均池化层，输出是一个二维张量。
  - `'max'` 代表全局最大池化
- **classes**: 可选，图片分类的类别数，仅当 `include_top` 为 `True` 并且不加载预训练权值时可用。

## 返回

一个 Keras Model 实例。

## 参考文献

- [Learning Transferable Architectures for Scalable Image Recognition](#)

## License

预训练权值基于 [Apache License](#)。

## MobileNetV2

```
keras.applications.mobilenet_v2.MobileNetV2(input_shape=None, alpha=1.0,
include_top=True, weights='imagenet', input_tensor=None, pooling=None,
classes=1000)
```

在 ImageNet 上预训练的 MobileNetV2 模型。

该模型可同时构建于 `channels_first` (通道, 高度, 宽度) 和 `channels_last` (高度, 宽度, 通道) 两种输入维度顺序。

模型默认输出尺寸为 224x224。

## 参数

- **`input_shape`**: 可选尺寸元组, 以确认你是否想使用一个输入图像像素不为 (224, 224, 3) 的模型。输入形状必须是 (224, 224, 3)。你也可以忽略这个选项, 如果你想从 `input_tensor` 来推断 `input_shape`。如果你选择同时包含 `input_tensor` 和 `input_shape`, 那么如果匹配的话会使用 `input_shape`, 如果不匹配会抛出错误。例如, (160, 160, 3) 是一个有效的值。
- **`alpha`**: 控制网络的宽度。这在 MobileNetV2 论文中被称作宽度乘子。
  - 如果 `alpha` < 1.0, 则同比例减少每层的滤波器个数。
  - 如果 `alpha` > 1.0, 则同比例增加每层的滤波器个数。
  - 如果 `alpha` = 1, 使用论文默认的滤波器个数。
- **`depth_multiplier`**: depthwise 卷积的深度乘子, 也称为 (分辨率乘子)
- **`include_top`**: 是否包括顶层的全连接层。
- **`weights`**: `None` 代表随机初始化, '`imagenet`' 代表加载在 ImageNet 上预训练的权值。
- **`input_tensor`**: 可选, Keras tensor 作为模型的输入 (即 `layers.Input()` 输出的 tensor)。
- **`pooling`**: 可选, 当 `include_top` 为 `False` 时, 该参数指定了特征提取时的池化方式。
  - `None` 代表不池化, 直接输出最后一层卷积块的输出, 该输出是一个 4D 张量。
  - '`avg`' 代表全局平均池化 (GlobalAveragePooling2D), 相当于在最后一层卷积块后面再加一层全局平均池化层, 输出是一个二维张量。
  - '`max`' 代表全局最大池化。
- **`classes`**: 可选, 图片分类的类别数, 仅当 `include_top` 为 `True` 并且不加载预训练权值时可用。

## 返回

一个 Keras `model` 实例。

## 异常

**`ValueError`**: 如果 `weights` 参数非法, 或非法的输入尺寸, 或者当 `weights='imagenet'` 时, 非法的 `alpha, rows`。

## 参考文献

- MobileNetV2: Inverted Residuals and Linear Bottlenecks

## License

预训练权值基于 [Apache License](#).



# Keras 后端

## 什么是「后端」？

Keras 是一个模型级库，为开发深度学习模型提供了高层次的构建模块。它不处理诸如张量乘积和卷积等低级操作。相反，它依赖于一个专门的、优化的张量操作库来完成这个操作，它可以作为 Keras 的「后端引擎」。相比单独地选择一个张量库，而将 Keras 的实现与该库相关联，Keras 以模块方式处理这个问题，并且可以将几个不同的后端引擎无缝嵌入到 Keras 中。

目前，Keras 有三个后端实现可用：[TensorFlow](#) 后端，[Theano](#) 后端，[CNTK](#) 后端。

- [TensorFlow](#) 是由 Google 开发的一个开源符号级张量操作框架。
- [Theano](#) 是由蒙特利尔大学的 LISA Lab 开发的一个开源符号级张量操作框架。
- [CNTK](#) 是由微软开发的一个深度学习开源工具包。

将来，我们可能会添加更多后端选项。

## 从一个后端切换到另一个后端

如果您至少运行过一次 Keras，您将在以下位置找到 Keras 配置文件：

```
$HOME/.keras/keras.json
```

如果它不在那里，你可以创建它。

**Windows 用户注意事项：**请将 `$HOME` 修改为 `%USERPROFILE%`。

默认的配置文件如下所示：

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

只需将字段 `backend` 更改为 `theano`，`tensorflow` 或 `cntk`，Keras 将在下次运行 Keras 代码时使用新的配置。

你也可以定义环境变量 `KERAS_BACKEND`，这会覆盖配置文件中定义的内容：

```
KERAS_BACKEND=tensorflow python -c "from keras import backend"
Using TensorFlow backend.
```

在 Keras 中，可以加载比 `"tensorflow"`, `"theano"` 和 `"cntk"` 更多的后端。Keras 也可以使用外部后端，这可以通过更改 `keras.json` 配置文件和 `"backend"` 设置来执行。假设您有一个名为 `my_module` 的 Python 模块，您希望将其用作外部后端。`keras.json` 配置文件将更改如下：

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

必须验证外部后端才能使用，有效的后端必须具有以下函数：`placeholder`, `variable` and `function`。

如果由于缺少必需的条目而导致外部后端无效，则会记录错误，通知缺少哪些条目。

## keras.json 详细配置

The `keras.json` 配置文件包含以下设置：

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

您可以通过编辑 `$ HOME/.keras/keras.json` 来更改这些设置。

- `image_data_format`: 字符串，`"channels_last"` 或者 `"channels_first"`。它指定了 Keras 将遵循的数据格式约定。`(keras.backend.image_data_format())` 返回它。
  - 对于 2D 数据（例如图像），`"channels_last"` 假定为 `(rows, cols, channels)`，而 `"channels_first"` 假定为 `(channels, rows, cols)`。
  - 对于 3D 数据，`"channels_last"` 假定为 `(conv_dim1, conv_dim2, conv_dim3, channels)`，而 `"channels_first"` 假定为 `(channels, conv_dim1, conv_dim2, conv_dim3)`。
- `epsilon`: 浮点数，用于避免在某些操作中被零除的数字模糊常量。
- `floatx`: 字符串，`"float16"`, `"float32"`, 或 `"float64"`。默认浮点精度。

- `backend`: 字符串，`"tensorflow"`, `"theano"`, 或 `"cntk"`。

## 使用抽象 Keras 后端编写新代码

如果你希望你编写的 Keras 模块与 Theano (`th`) 和 TensorFlow (`tf`) 兼容，则必须通过抽象 Keras 后端 API 来编写它们。以下是一个介绍。

您可以通过以下方式导入后端模块：

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

下面的代码实例化一个输入占位符。它等价于 `tf.placeholder()` 或 `th.tensor.matrix()`, `th.tensor.tensor3()`, 等等。

```
inputs = K.placeholder(shape=(2, 4, 5))
# 同样可以:
inputs = K.placeholder(shape=(None, 4, 5))
# 同样可以:
inputs = K.placeholder(ndim=3)
```

下面的代码实例化一个变量。它等价于 `tf.Variable()` 或 `th.shared()`。

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

# 全 0 变量:
var = K.zeros(shape=(3, 4, 5))
# 全 1 变量:
var = K.ones(shape=(3, 4, 5))
```

你需要的大多数张量操作都可以像在 TensorFlow 或 Theano 中那样完成：

```
# 使用随机数初始化张量
b = K.random_uniform_variable(shape=(3, 4), low=0, high=1) # 均匀分布
c = K.random_normal_variable(shape=(3, 4), mean=0, scale=1) # 高斯分布
d = K.random_normal_variable(shape=(3, 4), mean=0, scale=1)

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

## 后端函数

### backend

```
backend.backend()
```

返回当前后端的名字 (例如 “tensorflow”)。

#### 返回

字符串，Keras 目前正在使用的后端名。

#### 示例

```
>>> keras.backend.backend()  
'tensorflow'
```

## symbolic

```
keras.backend.symbolic(func)
```

在 TensorFlow 2.0 中用于进入 Keras 图的装饰器。

#### 参数

- **func**: 需要装饰的函数。

#### 返回

装饰后的函数。

## eager

```
keras.backend.eager(func)
```

在 TensorFlow 2.0 中用于退出 Keras 图的装饰器。

#### 参数

- **func**: 需要装饰的函数。

#### 返回

装饰后的函数。

## get\_uid

```
keras.backend.get_uid(prefix='')
```

提供一个戴字符串前缀的独立 UID。

### 参数

- **prefix**: 字符串。

### 返回

一个整数。

### 示例

```
>>> keras.backend.get_uid('dense')
1
>>> keras.backend.get_uid('dense')
2
```

## manual\_variable\_initialization

```
keras.backend.manual_variable_initialization(value)
```

设置手动变量初始化标识。

这个布尔标识决定变量是否在实例化（默认）时初始化，或者让用户自己来处理初始化。

### 参数

- **value**: Python 布尔值。

## epsilon

```
keras.backend.epsilon()
```

返回数字表达式中使用的模糊因子的值。

### 返回

一个浮点数。

## 示例

```
>>> keras.backend.epsilon()
1e-07
```

## reset\_uids

```
keras.backend.reset_uids()
```

重置图标识。

## set\_epsilon

```
keras.backend.set_epsilon(e)
```

设置数字表达式中使用的模糊因子的值。

## 参数

- **e**: 浮点数。新的 epsilon 值。

## 示例

```
>>> from keras import backend as K
>>> K.epsilon()
1e-07
>>> K.set_epsilon(1e-05)
>>> K.epsilon()
1e-05
```

## floatx

```
keras.backend.floatx()
```

以字符串形式返回默认的浮点类型。 (例如，'float16', 'float32', 'float64')。

## 返回

字符串，当前默认的浮点类型。

## 示例

```
>>> keras.backend.floatx()
'float32'
```

## set\_floatx

```
keras.backend.set_floatx(floatx)
```

设置默认的浮点类型。

## 参数

- **floatx**: 字符串，'float16', 'float32', 或 'float64'。

## 示例

```
>>> from keras import backend as K
>>> K.floatx()
'float32'
>>> K.set_floatx('float16')
>>> K.floatx()
'float16'
```

## cast\_to\_floatx

```
keras.backend.cast_to_floatx(x)
```

将 Numpy 数组转换为默认的 Keras 浮点类型。

## 参数

- **x**: Numpy 数组。

## 返回

相同的 Numpy 数组，转换为它的新类型。

## 示例

```
>>> from keras import backend as K
>>> K.floatx()
'float32'
>>> arr = numpy.array([1.0, 2.0], dtype='float64')
>>> arr.dtype
```

```
dtype('float64')
>>> new_arr = K.cast_to_floatx(arr)
>>> new_arr
array([ 1.,  2.], dtype=float32)
>>> new_arr.dtype
dtype('float32')
```

## image\_data\_format

```
keras.backend.image_data_format()
```

返回默认图像数据格式约定。

### 返回

一个字符串，`'channels_first'` 或 `'channels_last'`

## 示例

```
>>> keras.backend.image_data_format()
'channels_first'
```

## set\_image\_data\_format

```
keras.backend.set_image_data_format(data_format)
```

设置数据格式约定的值。

### 参数

- **data\_format**: 字符串。`'channels_first'` 或 `'channels_last'`。

## 示例

```
>>> from keras import backend as K
>>> K.image_data_format()
'channels_first'
>>> K.set_image_data_format('channels_last')
>>> K.image_data_format()
'channels_last'
```

## learning\_phase

```
keras.backend.learning_phase()
```

返回学习阶段的标志。

学习阶段标志是一个布尔张量（0 = test，1 = train），它作为输入传递给任何的 Keras 函数，以便在训练和测试时执行不同的行为操作。

## 返回

学习阶段 (标量整数张量或 python 整数)。

## set\_learning\_phase

```
keras.backend.set_learning_phase(value)
```

将学习阶段设置为固定值。

## 参数

- **value**: 学习阶段的值，0 或 1（整数）。

## 异常

- **ValueError**: 如果 value 既不是 0 也不是 1。

## clear\_session

```
keras.backend.clear_session()
```

销毁当前的 Keras 图并创建一个新图。

有用于避免旧模型/网络层混乱。

## is\_sparse

```
keras.backend.is_sparse(tensor)
```

判断张量是否是稀疏张量。

## 参数

- **tensor**: 一个张量实例。

## 返回

布尔值。

## 示例

```
>>> from keras import backend as K
>>> a = K.placeholder((2, 2), sparse=False)
>>> print(K.is_sparse(a))
False
>>> b = K.placeholder((2, 2), sparse=True)
>>> print(K.is_sparse(b))
True
```

## to\_dense

```
keras.backend.to_dense(tensor)
```

将稀疏张量转换为稠密张量并返回。

## 参数

- **tensor**: 张量实例（可能稀疏）。

## 返回

一个稠密张量。

## 示例

```
>>> from keras import backend as K
>>> b = K.placeholder((2, 2), sparse=True)
>>> print(K.is_sparse(b))
True
>>> c = K.to_dense(b)
>>> print(K.is_sparse(c))
False
```

## variable

```
keras.backend.variable(value, dtype=None, name=None, constraint=None)
```

实例化一个变量并返回它。

## 参数

- **value**: Numpy 数组，张量的初始值。
- **dtype**: 张量类型。
- **name**: 张量的可选名称字符串。
- **constraint**: 在优化器更新后应用于变量的可选投影函数。

## 返回

变量实例（包含 Keras 元数据）

## 示例

```
>>> from keras import backend as K
>>> val = np.array([[1, 2], [3, 4]])
>>> kvar = K.variable(value=val, dtype='float64', name='example_var')
>>> K.dtype(kvar)
'float64'
>>> print(kvar)
example_var
>>> K.eval(kvar)
array([[ 1.,  2.],
       [ 3.,  4.]])
```

## is\_variable

```
keras.backend.is_variable(x)
```

## constant

```
keras.backend.constant(value, dtype=None, shape=None, name=None)
```

创建一个常数张量。

## 参数

- **value**: 一个常数值（或列表）
- **dtype**: 结果张量的元素类型。
- **shape**: 可选的结果张量的尺寸。
- **name**: 可选的张量的名称。

## 返回

一个常数张量。

## is\_keras\_tensor

```
keras.backend.is_keras_tensor(x)
```

判断 `x` 是否是 Keras 张量

「Keras张量」是由 Keras 层（Layer 类）或 Input 返回的张量。

### 参数

- `x`: 候选张量。

### 返回

布尔值：参数是否是 Keras 张量。

### 异常

- **ValueError**: 如果 `x` 不是一个符号张量。

### 示例

```
>>> from keras import backend as K
>>> from keras.layers import Input, Dense
>>> np_var = numpy.array([1, 2])
>>> K.is_keras_tensor(np_var) # 一个 Numpy 数组不是一个符号张量。
ValueError
>>> k_var = tf.placeholder('float32', shape=(1, 1))
# 在 Keras 之外间接创建的变量不是 Keras 张量。
>>> K.is_keras_tensor(k_var)
False
>>> keras_var = K.variable(np_var)
# Keras 后端创建的变量不是 Keras 张量。
>>> K.is_keras_tensor(keras_var)
False
>>> keras_placeholder = K.placeholder(shape=(2, 4, 5))
# 占位符不是 Keras 张量。
>>> K.is_keras_tensor(keras_placeholder)
False
>>> keras_input = Input([10])
>>> K.is_keras_tensor(keras_input) # 输入 Input 是 Keras 张量。
True
>>> keras_layer_output = Dense(10)(keras_input)
# 任何 Keras 层输出都是 Keras 张量。
>>> K.is_keras_tensor(keras_layer_output)
True
```

## is\_tensor

```
keras.backend.is_tensor(x)
```

## placeholder

```
keras.backend.placeholder(shape=None, ndim=None, dtype=None, sparse=False, name=None)
```

实例化一个占位符张量并返回它。

### 参数

- **shape**: 占位符尺寸 (整数元组，可能包含 None 项)。
- **ndim**: 张量的轴数。 { shape , ndim } 至少一个需要被指定。如果两个都被指定，那么使用 shape 。
- **dtype**: 占位符类型。
- **sparse**: 布尔值，占位符是否应该有一个稀疏类型。
- **name**: 可选的占位符的名称字符串。

### 返回

张量实例（包括 Keras 元数据）。

## 示例

```
>>> from keras import backend as K
>>> input_ph = K.placeholder(shape=(2, 4, 5))
>>> input_ph._keras_shape
(2, 4, 5)
>>> input_ph
<tf.Tensor 'Placeholder_4:0' shape=(2, 4, 5) dtype=float32>
```

## is\_placeholder

```
keras.backend.is_placeholder(x)
```

判断 x 是否是占位符。

### 参数

- **x**: 候选占位符。

## 返回

布尔值。

## shape

```
keras.backend.shape(x)
```

返回张量或变量的符号尺寸。

### 参数

- **x**: 张量或变量。

## 返回

符号尺寸（它本身就是张量）。

## 示例

```
# TensorFlow 示例
>>> from keras import backend as K
>>> tf_session = K.get_session()
>>> val = np.array([[1, 2], [3, 4]])
>>> kvar = K.variable(value=val)
>>> inputs = keras.backend.placeholder(shape=(2, 4, 5))
>>> K.shape(kvar)
<tf.Tensor 'Shape_8:0' shape=(2,) dtype=int32>
>>> K.shape(inputs)
<tf.Tensor 'Shape_9:0' shape=(3,) dtype=int32>
# 要得到整数尺寸（相反，你可以使用 K.int_shape(x)）
>>> K.shape(kvar).eval(session=tf_session)
array([2, 2], dtype=int32)
>>> K.shape(inputs).eval(session=tf_session)
array([2, 4, 5], dtype=int32)
```

## int\_shape

```
keras.backend.int_shape(x)
```

返回张量或变量的尺寸，作为 int 或 None 项的元组。

### 参数

- **x**: 张量或变量。

## 返回

整数元组（或 None 项）。

## 示例

```
>>> from keras import backend as K
>>> inputs = K.placeholder(shape=(2, 4, 5))
>>> K.int_shape(inputs)
(2, 4, 5)
>>> val = np.array([[1, 2], [3, 4]])
>>> kvar = K.variable(value=val)
>>> K.int_shape(kvar)
(2, 2)
```

## Numpy 实现

```
def int_shape(x):
    return x.shape
```

## ndim

```
keras.backend.ndim(x)
```

以整数形式返回张量中的轴数。

## 参数

- **x**: 张量或变量。

## 返回

整数 (标量), 轴的数量。

## 示例

```
>>> from keras import backend as K
>>> inputs = K.placeholder(shape=(2, 4, 5))
>>> val = np.array([[1, 2], [3, 4]])
>>> kvar = K.variable(value=val)
>>> K.ndim(inputs)
3
>>> K.ndim(kvar)
2
```

## Numpy 实现

```
def ndim(x):
    return x.ndim
```

## size

```
keras.backend.size(x, name=None)
```

返回张量尺寸。

### 参数

- **x**: 张量或变量。
- **name**: 操作名称（可选）。

### 返回

张量尺寸

## 示例

```
>>> from keras import backend as K
>>> val = np.array([[1, 2], [3, 4]])
>>> kvar = K.variable(value=val)
>>> K.size(inputs)
<tf.Tensor: id=9, shape=(), dtype=int32, numpy=4>
```

## dtype

```
keras.backend.dtype(x)
```

以字符串形式返回 Keras 张量或变量的 dtype。

### 参数

- **x**: 张量或变量。

### 返回

字符串，`x` 的 dtype。

## 示例

```
>>> from keras import backend as K
>>> K.dtype(K.placeholder(shape=(2, 4, 5)))
```

```
'float32'  
>>> K.dtype(K.placeholder(shape=(2,4,5), dtype='float32'))  
'float32'  
>>> K.dtype(K.placeholder(shape=(2,4,5), dtype='float64'))  
'float64'  
# Keras 变量  
>>> kvar = K.variable(np.array([[1, 2], [3, 4]]))  

```

## Numpy 实现

```
def dtype(x):  
    return x.dtype.name
```

## eval

```
keras.backend.eval(x)
```

估计一个张量的值。

### 参数

- **x**: 张量。

### 返回

Numpy 数组。

### 示例

```
>>> from keras import backend as K  
>>> kvar = K.variable(np.array([[1, 2], [3, 4]]), dtype='float32')  
>>> K.eval(kvar)  
array([[ 1.,  2.],  
       [ 3.,  4.]], dtype=float32)
```

## Numpy 实现

```
def eval(x):  
    return x
```

## zeros

```
keras.backend.zeros(shape, dtype=None, name=None)
```

实例化一个全零变量并返回它。

### 参数

- **shape**: 整数元组，返回的Keras变量的尺寸。
- **dtype**: 字符串，返回的 Keras 变量的数据类型。
- **name**: 字符串，返回的 Keras 变量的名称。

### 返回

一个变量（包括 Keras 元数据），用 `0.0` 填充。请注意，如果 `shape` 是符号化的，我们不能返回一个变量，而会返回一个动态尺寸的张量。

### 示例

```
>>> from keras import backend as K
>>> kvar = K.zeros((3,4))
>>> K.eval(kvar)
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]], dtype=float32)
```

## Numpy 实现

```
def zeros(shape, dtype=floatx(), name=None):
    return np.zeros(shape, dtype=dtype)
```

## ones

```
keras.backend.ones(shape, dtype=None, name=None)
```

实例化一个全一变量并返回它。

### 参数

- **shape**: 整数元组，返回的Keras变量的尺寸。
- **dtype**: 字符串，返回的 Keras 变量的数据类型。
- **name**: 字符串，返回的 Keras 变量的名称。

### 返回

一个 Keras 变量，用 1.0 填充。请注意，如果 shape 是符号化的，我们不能返回一个变量，而会返回一个动态尺寸的张量。

## 示例

```
>>> from keras import backend as K
>>> kvar = K.ones((3,4))
>>> K.eval(kvar)
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]], dtype=float32)
```

## Numpy 实现

```
def ones(shape, dtype=floatx(), name=None):
    return np.ones(shape, dtype=dtype)
```

## eye

```
keras.backend.eye(size, dtype=None, name=None)
```

实例化一个单位矩阵并返回它。

## 参数

- **size**: 元组，行和列的数目。如果是整数，则为行数。
- **dtype**: 字符串，返回的 Keras 变量的数据类型。
- **name**: 字符串，返回的 Keras 变量的名称。

## 返回

Keras 变量，一个单位矩阵。

## 示例

```
>>> from keras import backend as K
>>> K.eval(K.eye(3))
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]], dtype=float32)
>>> K.eval(K.eye((2, 3)))
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.]], dtype=float32)
```

## Numpy 实现

```
def eye(size, dtype=None, name=None):
    if isinstance(size, (list, tuple)):
        n, m = size
    else:
        n, m = size, size
    return np.eye(n, m, dtype=dtype)
```

## zeros\_like

```
keras.backend.zeros_like(x, dtype=None, name=None)
```

实例化与另一个张量相同尺寸的全零变量。

### 参数

- **x**: Keras 变量或 Keras 张量。
- **dtype**: 字符串，返回的 Keras 变量的类型。如果为 None，则使用 x 的类型。
- **name**: 字符串，所创建的变量的名称。

### 返回

一个 Keras 变量，其形状为 x，用零填充。

### 示例

```
>>> from keras import backend as K
>>> kvar = K.variable(np.random.random((2,3)))
>>> kvar_zeros = K.zeros_like(kvar)
>>> K.eval(kvar_zeros)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]], dtype=float32)
```

### Numpy 实现

```
def zeros_like(x, dtype=floatx(), name=None):
    return np.zeros_like(x, dtype=dtype)
```

## ones\_like

```
keras.backend.ones_like(x, dtype=None, name=None)
```

实例化与另一个张量相同形状的全一变量。

## 参数

- **x**: Keras 变量或张量。
- **dtype**: 字符串，返回的 Keras 变量的类型。如果为 None，则使用 x 的类型。
- **name**: 字符串，所创建的变量的名称。

## 返回

一个 Keras 变量，其形状为 x，用一填充。

## 示例

```
>>> from keras import backend as K
>>> kvar = K.variable(np.random.random((2,3)))
>>> kvar_ones = K.ones_like(kvar)
>>> K.eval(kvar_ones)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]], dtype=float32)
```

## Numpy 实现

```
def ones_like(x, dtype=floatx(), name=None):
    return np.ones_like(x, dtype=dtype)
```

## identity

```
keras.backend.identity(x, name=None)
```

返回与输入张量相同内容的张量。

## 参数

- **x**: 输入张量。
- **name**: 字符串，所创建的变量的名称。

## 返回

一个相同尺寸、类型和内容的张量。

## random\_uniform\_variable

```
keras.backend.random_uniform_variable(shape, low, high, dtype=None, name=None,
seed=None)
```

使用从均匀分布中抽样出来的值来实例化变量。

## 参数

- **shape**: 整数元组，返回的 Keras 变量的尺寸。
- **low**: 浮点数，输出间隔的下界。
- **high**: 浮点数，输出间隔的上界。
- **dtype**: 字符串，返回的 Keras 变量的数据类型。
- **name**: 字符串，返回的 Keras 变量的名称。
- **seed**: 整数，随机种子。

## 返回

一个 Keras 变量，以抽取的样本填充。

## 示例

```
# TensorFlow 示例
>>> kvar = K.random_uniform_variable((2,3), 0, 1)
>>> kvar
<tensorflow.python.ops.variables.Variable object at 0x10ab40b10>
>>> K.eval(kvar)
array([[ 0.10940075,  0.10047495,  0.476143  ],
       [ 0.66137183,  0.00869417,  0.89220798]], dtype=float32)
```

## Numpy 实现

```
def random_uniform_variable(shape, low, high, dtype=None, name=None,
                           seed=None):
    return (high - low) * np.random.random(shape).astype(dtype) + low
```

## random\_normal\_variable

```
keras.backend.random_normal_variable(shape, mean, scale, dtype=None,
                                      name=None, seed=None)
```

使用从正态分布中抽取的值实例化一个变量。

## 参数

- **shape**: 整数元组，返回的Keras变量的尺寸。
- **mean**: 浮点型，正态分布平均值。
- **scale**: 浮点型，正态分布标准差。

- **dtype**: 字符串，返回的Keras变量的 dtype。
- **name**: 字符串，返回的Keras变量的名称。
- **seed**: 整数，随机种子。

## 返回

一个 Keras 变量，以抽取的样本填充。

## 示例

```
# TensorFlow 示例
>>> kvar = K.random_normal_variable((2,3), 0, 1)
>>> kvar
<tensorflow.python.ops.variables.Variable object at 0x10ab12dd0>
>>> K.eval(kvar)
array([[ 1.19591331,  0.68685907, -0.63814116],
       [ 0.92629528,  0.28055015,  1.70484698]], dtype=float32)
```

## Numpy 实现

```
def random_normal_variable(shape, mean, scale, dtype=None, name=None,
                           seed=None):
    return scale * np.random.randn(*shape).astype(dtype) + mean
```

## count\_params

```
keras.backend.count_params(x)
```

返回 Keras 变量或张量中的静态元素数。

## 参数

- **x**: Keras 变量或张量。

## 返回

整数，`x` 中的元素数量，即，数组中静态维度的乘积。

## 示例

```
>>> kvar = K.zeros((2,3))
>>> K.count_params(kvar)
6
>>> K.eval(kvar)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]], dtype=float32)
```

## Numpy 实现

```
def count_params(x):
    return x.size
```

## cast

```
keras.backend.cast(x, dtype)
```

将张量转换到不同的 `dtype` 并返回。

你可以转换一个 Keras 变量，但它仍然返回一个 Keras 张量。

### 参数

- `x`: Keras 张量（或变量）。
- `dtype`: 字符串，(`'float16'`, `'float32'` 或 `'float64'`)。

### 返回

Keras 张量，类型为 `dtype`。

### 示例

```
>>> from keras import backend as K
>>> input = K.placeholder((2, 3), dtype='float32')
>>> input
<tf.Tensor 'Placeholder_2:0' shape=(2, 3) dtype=float32>
# It doesn't work in-place as below.
>>> K.cast(input, dtype='float16')
<tf.Tensor 'Cast_1:0' shape=(2, 3) dtype=float16>
>>> input
<tf.Tensor 'Placeholder_2:0' shape=(2, 3) dtype=float32>
# you need to assign it.
>>> input = K.cast(input, dtype='float16')
>>> input
<tf.Tensor 'Cast_2:0' shape=(2, 3) dtype=float16>
```

## update

```
keras.backend.update(x, new_x)
```

将 `x` 的值更新为 `new_x`。

## 参数

- **x**: 一个 `Variable`。
- **new\_x**: 一个与 `x` 尺寸相同的张量。

## 返回

更新后的变量 `x`。

## update\_add

```
keras.backend.update_add(x, increment)
```

通过增加 `increment` 来更新 `x` 的值。

## 参数

- **x**: 一个 `Variable`。
- **increment**: 与 `x` 形状相同的张量。

## 返回

更新后的变量 `x`。

## update\_sub

```
keras.backend.update_sub(x, decrement)
```

通过减 `decrement` 来更新 `x` 的值。

## 参数

- **x**: 一个 `Variable`。
- **decrement**: 与 `x` 形状相同的张量。

## 返回

更新后的变量 `x`。

## moving\_average\_update

```
keras.backend.moving_average_update(x, value, momentum)
```

计算变量的移动平均值。

## 参数

- **x**: 一个 Variable。
- **value**: 与 `x` 形状相同的张量。
- **momentum**: 移动平均动量。

## 返回

更新变量的操作。

## dot

```
keras.backend.dot(x, y)
```

将 2 个张量（和/或变量）相乘并返回一个\*张量\*。

当试图将 nD 张量与 nD 张量相乘时， 它会重现 Theano 行为。 (例如 `(2, 3) * (4, 3, 5) -> (2, 4, 5)`)

## 参数

- **x**: 张量或变量。
- **y**: 张量或变量。

## 返回

一个张量，`x` 和 `y` 的点积。

## 示例

```
# 张量之间的点积
>>> x = K.placeholder(shape=(2, 3))
>>> y = K.placeholder(shape=(3, 4))
>>> xy = K.dot(x, y)
>>> xy
<tf.Tensor 'MatMul_9:0' shape=(2, 4) dtype=float32>
```

```
# 张量之间的点积
>>> x = K.placeholder(shape=(32, 28, 3))
>>> y = K.placeholder(shape=(3, 4))
>>> xy = K.dot(x, y)
```

```
>>> xy
<tf.Tensor 'MatMul_9:0' shape=(32, 28, 4) dtype=float32>
```

```
# 类 Theano 行为的示例
>>> x = K.random_uniform_variable(shape=(2, 3), low=0, high=1)
>>> y = K.ones((4, 3, 5))
>>> xy = K.dot(x, y)
>>> K.int_shape(xy)
(2, 4, 5)
```

## Numpy 实现

```
def dot(x, y):
    return np.dot(x, y)
```

## batch\_dot

```
keras.backend.batch_dot(x, y, axes=None)
```

批量化的点积。

当 `x` 和 `y` 是批量数据时，`batch_dot` 用于计算 `x` 和 `y` 的点积，即尺寸为 `(batch_size, :)`。

`batch_dot` 产生一个比输入尺寸更小的张量或变量。如果维数减少到 1，我们使用 `expand_dims` 来确保 `ndim` 至少为 2。

### 参数

- `x`: `ndim >= 2` 的 Keras 张量或变量。
- `y`: `ndim >= 2` 的 Keras 张量或变量。
- `axes`: 整数或元组 (int, int)。需要归约的目标维度。

### 返回

一个尺寸等于 `x` 的尺寸（减去总和的维度）和 `y` 的尺寸（减去批次维度和总和的维度）的连接的张量。如果最后的秩为 1，我们将它重新转换为 `(batch_size, 1)`。

### 示例

假设 `x = [[1, 2], [3, 4]]` 和 `y = [[5, 6], [7, 8]]`，`batch_dot(x, y, axes=1) = [[17], [53]]` 是 `x.dot(y.T)` 的主对角线，尽管我们不需要计算非对角元素。

伪代码：

```
inner_products = []
for xi, yi in zip(x, y):
    inner_products.append(xi.dot(yi))
result = stack(inner_products)
```

尺寸推断：让 `x` 的尺寸为 `(100, 20)`，以及 `y` 的尺寸为 `(100, 30, 20)`。如果 `axes` 是 `(1, 2)`，要找出结果张量的尺寸，循环 `x` 和 `y` 的尺寸的每一个维度。

- `x.shape[0] : 100`：附加到输出形状，
- `x.shape[1] : 20`：不附加到输出形状，`x` 的第一个维度已经被加和了 (`dot_axes[0] = 1`)。
- `y.shape[0] : 100`：不附加到输出形状，总是忽略 `y` 的第一维
- `y.shape[1] : 30`：附加到输出形状，
- `y.shape[2] : 20`：不附加到输出形状，`y` 的第二个维度已经被加和了 (`dot_axes[0] = 2`)。`output_shape = (100, 30)`

```
>>> x_batch = K.ones(shape=(32, 20, 1))
>>> y_batch = K.ones(shape=(32, 30, 20))
>>> xy_batch_dot = K.batch_dot(x_batch, y_batch, axes=(1, 2))
>>> K.int_shape(xy_batch_dot)
(32, 1, 30)
```

## Numpy 实现

## 展示 Numpy 实现

```
def batch_dot(x, y, axes=None):
    if x.ndim < 2 or y.ndim < 2:
        raise ValueError('Batch dot requires inputs of rank 2 or more.')

    if isinstance(axes, int):
        axes = [axes, axes]
    elif isinstance(axes, tuple):
        axes = list(axes)

    if axes is None:
        if y.ndim == 2:
            axes = [x.ndim - 1, y.ndim - 1]
        else:
            axes = [x.ndim - 1, y.ndim - 2]

    if any([isinstance(a, (list, tuple)) for a in axes]):
        raise ValueError('Multiple target dimensions are not supported. ' +
                         'Expected: None, int, (int, int), ' +
                         'Provided: ' + str(axes))

    # 处理负轴
    if axes[0] < 0:
        axes[0] += x.ndim
    if axes[1] < 0:
        axes[1] += y.ndim

    if 0 in axes:
        raise ValueError('Can not perform batch dot over axis 0.')

    if x.shape[0] != y.shape[0]:
        raise ValueError('Can not perform batch dot on inputs' +
                         ' with different batch sizes.')

    d1 = x.shape[axes[0]]
    d2 = y.shape[axes[1]]
    if d1 != d2:
        raise ValueError('Can not do batch_dot on inputs with shapes ' +
                         str(x.shape) + ' and ' + str(y.shape) +
                         ' with axes=' + str(axes) + '. x.shape[%d] != ' +
                         'y.shape[%d] (%d != %d). ' % (axes[0], axes[1], d1,
d2))

    result = []
    axes = [axes[0] - 1, axes[1] - 1] # 忽略批次维度
    for xi, yi in zip(x, y):
        result.append(np.tensordot(xi, yi, axes))
    result = np.array(result)

    if result.ndim == 1:
        result = np.expand_dims(result, -1)

    return result
```

## transpose

```
keras.backend.transpose(x)
```

将张量转置并返回。

### 参数

- **x**: 张量或变量。

### 返回

一个张量。

### 示例

```
>>> var = K.variable([[1, 2, 3], [4, 5, 6]])
>>> K.eval(var)
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]], dtype=float32)
>>> var_transposed = K.transpose(var)
>>> K.eval(var_transposed)
array([[ 1.,  4.],
       [ 2.,  5.],
       [ 3.,  6.]], dtype=float32)
```

```
>>> inputs = K.placeholder((2, 3))
>>> inputs
<tf.Tensor 'Placeholder_11:0' shape=(2, 3) dtype=float32>
>>> input_transposed = K.transpose(inputs)
>>> input_transposed
<tf.Tensor 'transpose_4:0' shape=(3, 2) dtype=float32>
```

## Numpy 实现

```
def transpose(x):
    return np.transpose(x)
```

## gather

```
keras.backend.gather(reference, indices)
```

在张量 `reference` 中检索索引 `indices` 的元素。

### 参数

- **reference**: 一个张量。

- **indices**: 索引的整数张量。

## 返回

与 `reference` 类型相同的张量。

## Numpy 实现

```
def gather(reference, indices):
    return reference[indices]
```

## max

```
keras.backend.max(x, axis=None, keepdims=False)
```

张量中的最大值。

## 参数

- **x**: 张量或变量。
- **axis**: 一个整数，需要在哪个轴寻找最大值。
- **keepdims**: 布尔值，是否保留原尺寸。如果 `keepdims` 为 `False`，则张量的秩减 1。如果 `keepdims` 为 `True`，缩小的维度保留为长度 1。

## 返回

`x` 中最大值的张量。

## Numpy 实现

```
def max(x, axis=None, keepdims=False):
    if isinstance(axis, list):
        axis = tuple(axis)
    return np.max(x, axis=axis, keepdims=keepdims)
```

## min

```
keras.backend.min(x, axis=None, keepdims=False)
```

张量中的最小值。

## 参数

- **x**: 张量或变量。
- **axis**: 一个整数，需要在哪个轴寻找最大值。
- **keepdims**: 布尔值，是否保留原尺寸。如果 `keepdims` 为 `False`，则张量的秩减 1。如果 `keepdims` 为 `True`，缩小的维度保留为长度 1。

## 返回

`x` 中最小值的张量。

## Numpy 实现

```
def min(x, axis=None, keepdims=False):
    if isinstance(axis, list):
        axis = tuple(axis)
    return np.min(x, axis=axis, keepdims=keepdims)
```

## sum

```
keras.backend.sum(x, axis=None, keepdims=False)
```

计算张量在某一指定轴的和。

## 参数

- **x**: 张量或变量。
- **axis**: 一个整数，需要加和的轴。
- **keepdims**: 布尔值，是否保留原尺寸。如果 `keepdims` 为 `False`，则张量的秩减 1。如果 `keepdims` 为 `True`，缩小的维度保留为长度 1。

## 返回

`x` 的和的张量。

## Numpy 实现

```
def sum(x, axis=None, keepdims=False):
    if isinstance(axis, list):
        axis = tuple(axis)
    return np.sum(x, axis=axis, keepdims=keepdims)
```

## prod

```
keras.backend.prod(x, axis=None, keepdims=False)
```

在某一指定轴，计算张量中的值的乘积。

### 参数

- **x**: 张量或变量。
- **axis**: 一个整数需要计算乘积的轴。
- **keepdims**: 布尔值，是否保留原尺寸。如果 `keepdims` 为 `False`，则张量的秩减 1。如果 `keepdims` 为 `True`，缩小的维度保留为长度 1。

### 返回

`x` 的元素的乘积的张量。

## Numpy 实现

```
def prod(x, axis=None, keepdims=False):
    if isinstance(axis, list):
        axis = tuple(axis)
    return np.prod(x, axis=axis, keepdims=keepdims)
```

## cumsum

```
keras.backend.cumsum(x, axis=0)
```

在某一指定轴，计算张量中的值的累加和。

### 参数

- **x**: 张量或变量。
- **axis**: 一个整数，需要加和的轴。

### 返回

`x` 在 `axis` 轴的累加和的张量。

## Numpy 实现

```
def cumsum(x, axis=0):
    return np.cumsum(x, axis=axis)
```

## cumprod

```
keras.backend.cumprod(x, axis=0)
```

在某一指定轴，计算张量中的值的累积乘积。

### 参数

- **x**: 张量或变量。
- **axis**: 一个整数，需要计算乘积的轴。

### 返回

`x` 在 `axis` 轴的累乘的张量。

## Numpy 实现

```
def cumprod(x, axis=0):  
    return np.cumprod(x, axis=axis)
```

## var

```
keras.backend.var(x, axis=None, keepdims=False)
```

张量在某一指定轴的方差。

### 参数

- **x**: 张量或变量。
- **axis**: 一个整数，要计算方差的轴。
- **keepdims**: 布尔值，是否保留原尺寸。如果 `keepdims` 为 `False`，则张量的秩减 1。如果 `keepdims` 为 `True`，缩小的维度保留为长度 1。

### 返回

`x` 元素的方差的张量。

## Numpy 实现

```
def var(x, axis=None, keepdims=False):  
    if isinstance(axis, list):  
        axis = tuple(axis)  
    return np.var(x, axis=axis, keepdims=keepdims)
```

## std

```
keras.backend.std(x, axis=None, keepdims=False)
```

张量在某一指定轴的标准差。

### 参数

- **x**: 张量或变量。
- **axis**: 一个整数，要计算标准差的轴。
- **keepdims**: 布尔值，是否保留原尺寸。如果 `keepdims` 为 `False`，则张量的秩减 1。如果 `keepdims` 为 `True`，缩小的维度保留为长度 1。

### 返回

`x` 元素的标准差的张量。

## Numpy 实现

```
def std(x, axis=None, keepdims=False):
    if isinstance(axis, list):
        axis = tuple(axis)
    return np.std(x, axis=axis, keepdims=keepdims)
```

## mean

```
keras.backend.mean(x, axis=None, keepdims=False)
```

张量在某一指定轴的均值。

### 参数

- **x**: A tensor or variable.
- **axis**: 整数或列表。需要计算均值的轴。
- **keepdims**: 布尔值，是否保留原尺寸。如果 `keepdims` 为 `False`，则 `axis` 中每一项的张量秩减 1。如果 `keepdims` 为 `True`，则缩小的维度保留为长度 1。

### 返回

`x` 元素的均值的张量。

## Numpy 实现

```
def mean(x, axis=None, keepdims=False):
    if isinstance(axis, list):
        axis = tuple(axis)
    return np.mean(x, axis=axis, keepdims=keepdims)
```

## any

```
keras.backend.any(x, axis=None, keepdims=False)
```

### reduction

按位归约（逻辑 OR）。

#### 参数

- **x**: 张量或变量。
- **axis**: 执行归约操作的轴。
- **keepdims**: 是否放弃或广播归约的轴。

#### 返回

一个 uint8 张量 (0s 和 1s)。

## Numpy 实现

```
def any(x, axis=None, keepdims=False):
    if isinstance(axis, list):
        axis = tuple(axis)
    return np.any(x, axis=axis, keepdims=keepdims)
```

## all

```
keras.backend.all(x, axis=None, keepdims=False)
```

按位归约（逻辑 AND）。

#### 参数

- **x**: 张量或变量。
- **axis**: 执行归约操作的轴。
- **keepdims**: 是否放弃或广播归约的轴。

## 返回

一个 uint8 张量 (0s 和 1s)。

## Numpy 实现

```
def all(x, axis=None, keepdims=False):
    if isinstance(axis, list):
        axis = tuple(axis)
    return np.all(x, axis=axis, keepdims=keepdims)
```

## argmax

```
keras.backend.argmax(x, axis=-1)
```

返回指定轴的最大值的索引。

## 参数

- **x**: 张量或变量。
- **axis**: 执行归约操作的轴。

## 返回

一个张量。

## Numpy 实现

```
def argmax(x, axis=-1):
    return np.argmax(x, axis=axis)
```

## argmin

```
keras.backend.argmin(x, axis=-1)
```

返回指定轴的最小值的索引。

## 参数

- **x**: 张量或变量。
- **axis**: 执行归约操作的轴。

## 返回

一个张量。

## Numpy 实现

```
def argmin(x, axis=-1):
    return np.argmin(x, axis=axis)
```

## square

```
keras.backend.square(x)
```

元素级的平方操作。

### 参数

- **x**: 张量或变量。

### 返回

一个张量。

## abs

```
keras.backend.abs(x)
```

元素级的绝对值操作。

### 参数

- **x**: 张量或变量。

### 返回

一个张量。

## sqrt

```
keras.backend.sqrt(x)
```

元素级的平方根操作。

## 参数

- **x**: 张量或变量。

## 返回

一个张量。

## Numpy 实现

```
def sqrt(x):  
    y = np.sqrt(x)  
    y[np.isnan(y)] = 0.  
    return y
```

## exp

```
keras.backend.exp(x)
```

元素级的指数运算操作。

## 参数

- **x**: 张量或变量。

## 返回

一个张量。

## log

```
keras.backend.log(x)
```

元素级的对数运算操作。

## 参数

- **x**: 张量或变量。

## 返回

一个张量。

## logsumexp

```
keras.backend.logsumexp(x, axis=None, keepdims=False)
```

计算  $\log(\sum(\exp(\text{张量在某一轴的元素})))$ 。

这个函数在数值上比  $\log(\sum(\exp(x)))$  更稳定。它避免了求大输入的指数造成的上溢，以及求小输入的对数造成的下溢。

### 参数

- **x**: 张量或变量。
- **axis**: 一个整数，需要归约的轴。
- **keepdims**: 布尔值，是否保留原尺寸。如果 `keepdims` 为 `False`，则张量的秩减 1。如果 `keepdims` 为 `True`，缩小的维度保留为长度 1。

### 返回

归约后的张量。

## Numpy 实现

```
def logsumexp(x, axis=None, keepdims=False):
    if isinstance(axis, list):
        axis = tuple(axis)
    return sp.special.logsumexp(x, axis=axis, keepdims=keepdims)
```

## round

```
keras.backend.round(x)
```

元素级地四舍五入到最接近的整数。

在平局的情况下，使用的舍入模式是「偶数的一半」。

### 参数

- **x**: 张量或变量。

### 返回

一个张量。

## sign

```
keras.backend.sign(x)
```

元素级的符号运算。

### 参数

- **x**: 张量或变量。

### 返回

一个张量。

## pow

```
keras.backend.pow(x, a)
```

元素级的指数运算操作。

### 参数

- **x**: 张量或变量。
- **a**: Python 整数。

### 返回

一个张量。

## Numpy 实现

```
def pow(x, a=1.):
    return np.power(x, a)
```

## clip

```
keras.backend.clip(x, min_value, max_value)
```

元素级裁剪。

### 参数

- **x**: 张量或变量。

- **min\_value**: Python 浮点，整数或张量。
- **max\_value**: Python 浮点，整数或张量。

## 返回

一个张量。

## Numpy 实现

```
def clip(x, min_value, max_value):  
    return np.clip(x, min_value, max_value)
```

## equal

```
keras.backend.equal(x, y)
```

逐个元素对比两个张量的相等情况。

## 参数

- **x**: 张量或变量。
- **y**: 张量或变量。

## 返回

一个布尔张量。

## Numpy 实现

```
def equal(x, y):  
    return x == y
```

## not\_equal

```
keras.backend.not_equal(x, y)
```

逐个元素对比两个张量的不相等情况。

## 参数

- **x**: 张量或变量。
- **y**: 张量或变量。

## 返回

一个布尔张量。

## Numpy 实现

```
def not_equal(x, y):  
    return x != y
```

## greater

```
keras.backend.greater(x, y)
```

逐个元素比对 ( $x > y$ ) 的真值。

## 参数

- **x**: 张量或变量。
- **y**: 张量或变量。

## 返回

一个布尔张量。

## Numpy 实现

```
def greater(x, y):  
    return x > y
```

## greater\_equal

```
keras.backend.greater_equal(x, y)
```

逐个元素比对 ( $x \geq y$ ) 的真值。

## 参数

- **x**: 张量或变量。
- **y**: 张量或变量。

## 返回

一个布尔张量。

## Numpy 实现

```
def greater_equal(x, y):  
    return x >= y
```

## less

```
keras.backend.less(x, y)
```

逐个元素比对 ( $x < y$ ) 的真值。

### 参数

- **x**: 张量或变量。
- **y**: 张量或变量。

### 返回

一个布尔张量。

## Numpy 实现

```
def less(x, y):  
    return x < y
```

## less\_equal

```
keras.backend.less_equal(x, y)
```

逐个元素比对 ( $x \leq y$ ) 的真值。

### 参数

- **x**: 张量或变量。
- **y**: 张量或变量。

### 返回

一个布尔张量。

## Numpy 实现

```
def less_equal(x, y):
    return x <= y
```

## maximum

```
keras.backend.maximum(x, y)
```

逐个元素比对两个张量的最大值。

### 参数

- **x**: 张量或变量。
- **y**: 张量或变量。

### 返回

一个张量。

### Numpy 实现

```
def maximum(x, y):
    return np.maximum(x, y)
```

## minimum

```
keras.backend.minimum(x, y)
```

逐个元素比对两个张量的最小值。

### 参数

- **x**: 张量或变量。
- **y**: 张量或变量。

### 返回

一个张量。

### Numpy 实现

```
def minimum(x, y):
    return np.minimum(x, y)
```

## sin

```
keras.backend.sin(x)
```

逐个元素计算  $x$  的 sin 值。

### 参数

- **x**: 张量或变量。

### 返回

一个张量。

## cos

```
keras.backend.cos(x)
```

逐个元素计算  $x$  的 cos 值。

### 参数

- **x**: 张量或变量。

### 返回

一个张量。

## normalize\_batch\_in\_training

```
keras.backend.normalize_batch_in_training(x, gamma, beta, reduction_axes,  
epsilon=0.001)
```

计算批次的均值和标准差，然后在批次上应用批次标准化。

### 参数

- **x**: 输入张量或变量。
- **gamma**: 用于缩放输入的张量。
- **beta**: 用于中心化输入的张量。
- **reduction\_axes**: 整数迭代，需要标准化的轴。
- **epsilon**: 模糊因子。

## 返回

长度为 3 个元组，`(normalized_tensor, mean, variance)`。

## batch\_normalization

```
keras.backend.batch_normalization(x, mean, var, beta, gamma, epsilon=0.001)
```

在给定的 `mean`, `var`, `beta` 和 `gamma` 上应用批量标准化。

即，返回：`output = (x - mean) / (sqrt(var) + epsilon) * gamma + beta`

## 参数

- **x**: 输入张量或变量。
- **mean**: 批次的均值。
- **var**: 批次的方差。
- **beta**: 用于中心化输入的张量。
- **gamma**: 用于缩放输入的张量。
- **epsilon**: 模糊因子。

## 返回

一个张量。

## concatenate

```
keras.backend.concatenate(tensors, axis=-1)
```

基于指定的轴，连接张量的列表。

## 参数

- **tensors**: 需要连接的张量列表。
- **axis**: 连接的轴。

## 返回

一个张量。

## reshape

```
keras.backend.reshape(x, shape)
```

将张量重塑为指定的尺寸。

### 参数

- **x**: 张量或变量。
- **shape**: 目标尺寸元组。

### 返回

一个张量。

## permute\_dimensions

```
keras.backend.permute_dimensions(x, pattern)
```

重新排列张量的轴。

### 参数

- **x**: 张量或变量。
- **pattern**: 维度索引的元组，例如 (0, 2, 1)。

### 返回

一个张量。

## resize\_images

```
keras.backend.resize_images(x, height_factor, width_factor, data_format)
```

调整 4D 张量中包含的图像的大小。

### 参数

- **x**: 需要调整的张量或变量。
- **height\_factor**: 正整数。
- **width\_factor**: 正整数。

- **data\_format**: 字符串， "channels\_last" 或 "channels\_first"。

## 返回

一个张量。

## 异常

- **ValueError**: 如果 `data_format` 既不是 "channels\_last" 也不是 "channels\_first"。

## resize\_volumes

```
keras.backend.resize_volumes(x, depth_factor, height_factor, width_factor,  
data_format)
```

调整 5D 张量中包含的体积。

## 参数

- **x**: 需要调整的张量或变量。
- **depth\_factor**: 正整数。
- **height\_factor**: 正整数。
- **width\_factor**: 正整数。
- **data\_format**: 字符串， "channels\_last" 或 "channels\_first"。

## 返回

一个张量。

## 异常

- **ValueError**: 如果 `data_format` 既不是 "channels\_last" 也不是 "channels\_first"。

## repeat\_elements

```
keras.backend.repeat_elements(x, rep, axis)
```

沿某一轴重复张量的元素，如 `np.repeat`。

如果 `x` 的尺寸为 `(s1, s2, s3)` 而 `axis` 为 `1`，则输出尺寸为 `(s1, s2 * rep, s3)`。

## 参数

- **x**: 张量或变量。
- **rep**: Python 整数，重复次数。
- **axis**: 需要重复的轴。

## 返回

一个张量。

## repeat

```
keras.backend.repeat(x, n)
```

重复一个 2D 张量。

如果 `x` 的尺寸为 `(samples, dim)` 并且 `n` 为 2，则输出的尺寸为 `(samples, 2, dim)`。

## 参数

- **x**: 张量或变量。
- **n**: Python 整数，重复次数。

## 返回

一个张量。

## arange

```
keras.backend.arange(start, stop=None, step=1, dtype='int32')
```

创建一个包含整数序列的 1D 张量。

该函数参数与 Theano 的 `arange` 函数的约定相同：如果只提供了一个参数，那它就是 `stop` 参数。

返回的张量的默认类型是 `int32`，以匹配 TensorFlow 的默认值。

## 参数

- **start**: 起始值。
- **stop**: 结束值。

- **step**: 两个连续值之间的差。

- **dtype**: 要使用的整数类型。

## 返回

一个整数张量。

## tile

```
keras.backend.tile(x, n)
```

创建一个用 `n` 平铺的 `x` 张量。

## 参数

- **x**: 张量或变量。

- **n**: 整数列表。长度必须与 `x` 中的维数相同。

## 返回

一个平铺的张量。

## 示例

```
>>> from keras import backend as K
>>> kvar = K.variable(np.random.random((2, 3)))
>>> kvar_tile = K.tile(K.eye(2), (2, 3))
>>> K.eval(kvar_tile)
array([[1.,  0.,  1.,  0.,  1.,  0.],
       [0.,  1.,  0.,  1.,  0.,  1.],
       [1.,  0.,  1.,  0.,  1.,  0.],
       [0.,  1.,  0.,  1.,  0.,  1.]], dtype=float32)
```

## Numpy 实现

```
def tile(x, n):
    return np.tile(x, n)
```

## flatten

```
keras.backend.flatten(x)
```

展平一个张量。

## 参数

- **x**: 张量或变量。

## 返回

一个重新调整为 1D 的张量。

## batch\_flatten

```
keras.backend.batch_flatten(x)
```

将一个 nD 张量变成一个 第 0 维相同的 2D 张量。

换句话说，它将批次中的每一个样本展平。

## 参数

- **x**: 张量或变量。

## 返回

一个张量。

## expand\_dims

```
keras.backend.expand_dims(x, axis=-1)
```

在索引 `axis` 轴，添加 1 个尺寸的维度。

## 参数

- **x**: 张量或变量。
- **axis**: 需要添加新的轴的位置。

## 返回

一个扩展维度的轴。

## squeeze

```
keras.backend.squeeze(x, axis)
```

在索引 `axis` 轴，移除 1 个尺寸的维度。

## 参数

- `x`: 张量或变量。
- `axis`: 需要丢弃的轴。

## 返回

一个与 `x` 数据相同但维度降低的张量。

## temporal\_padding

```
keras.backend.temporal_padding(x, padding=(1, 1))
```

填充 3D 张量的中间维度。

## 参数

- `x`: 张量或变量。
- `padding`: 2 个整数的元组，在第一个维度的开始和结束处添加多少个零。 [返回](#)

一个填充的 3D 张量。

## spatial\_2d\_padding

```
keras.backend.spatial_2d_padding(x, padding=((1, 1), (1, 1)),  
data_format=None)
```

填充 4D 张量的第二维和第三维。

## 参数

- `x`: 张量或变量。
- `padding`: 2 元组的元组，填充模式。
- `data_format`: 字符串，`"channels_last"` 或 `"channels_first"`。

## 返回

一个填充的 4D 张量。

## 异常

- **ValueError**: 如果 `data_format` 既不是 `"channels_last"` 也不是 `"channels_first"`。

## spatial\_3d\_padding

```
keras.backend.spatial_3d_padding(x, padding=((1, 1), (1, 1), (1, 1)),  
data_format=None)
```

沿着深度、高度宽度三个维度填充 5D 张量。

分别使用 “padding[0]”, “padding[1]” 和 “padding[2]” 来左右填充这些维度。

对于 ‘channels\_last’ 数据格式， 第 2、3、4 维将被填充。对于 ‘channels\_first’ 数据格式， 第 3、4、5 维将被填充。

## 参数

- **x**: 张量或变量。
- **padding**: 3 元组的元组， 填充模式。
- **data\_format**: 字符串， `"channels_last"` 或 `"channels_first"`。

## 返回

一个填充的 5D 张量。

## 异常

- **ValueError**: 如果 `data_format` 既不是 `"channels_last"` 也不是 `"channels_first"`。

## stack

```
keras.backend.stack(x, axis=0)
```

将秩为 `R` 的张量列表堆叠成秩为 `R + 1` 的张量。

## 参数

- **x**: 张量列表。
- **axis**: 需要执行堆叠的轴。

## 返回

一个张量。

## Numpy 实现

```
def stack(x, axis=0):
    return np.stack(x, axis=axis)
```

## one\_hot

```
keras.backend.one_hot(indices, num_classes)
```

计算一个整数张量的 one-hot 表示。

### 参数

- **indices**: nD 整数，尺寸为 (batch\_size, dim1, dim2, ... dim(n-1))
- **num\_classes**: 整数，需要考虑的类别数。

### 返回

输入的 (n + 1)D one-hot 表示，尺寸为 (batch\_size, dim1, dim2, ... dim(n-1), num\_classes)。

## reverse

```
keras.backend.reverse(x, axes)
```

沿指定的轴反转张量。

### 参数

- **x**: 需要反转的张量。
- **axes**: 整数或整数迭代。需要反转的轴。

### 返回

一个张量。

## Numpy 实现

```
def reverse(x, axes):
    if isinstance(axes, list):
```

```
    axes = tuple(axes)
    return np.flip(x, axes)
```

## slice

```
keras.backend.slice(x, start, size)
```

从张量中提取一个切片。

### 参数

- **x**: 输入张量。
- **start**: 整数列表/元组，表明每个轴的起始切片索引位置。
- **size**: 整数列表/元组，表明每个轴上切片多少维度。

### 返回

一个切片张量：

```
new_x = x[start[0]: start[0] + size[0], ..., start[-1]: start[-1] + size[-1]]
```

### 异常

- **ValueError**: 如果维度和索引的尺寸不匹配。

## Numpy 实现

```
def slice(x, start, size):
    slices = [py_slice(i, i + j) for i, j in zip(start, size)]
    return x[tuple(slices)]
```

## get\_value

```
keras.backend.get_value(x)
```

返回一个变量的值。

### 参数

- **x**: 输入变量。

### 返回

一个 Numpy 数组。

## batch\_get\_value

```
keras.backend.batch_get_value(ops)
```

返回多个张量变量的值。

### 参数

- **ops**: 要运行的操作列表。

### 返回

一个 Numpy 数组的列表。

## set\_value

```
keras.backend.set_value(x, value)
```

使用 Numpy 数组设置变量的值。

### 参数

- **x**: 需要设置新值的变量。
- **value**: 需要设置的值，一个尺寸相同的 Numpy 数组。

## batch\_set\_value

```
keras.backend.batch_set_value(tuples)
```

一次设置多个张量变量的值。

### 参数

- **tuples**: 元组 (tensor, value) 的列表。 value 应该是一个 Numpy 数组。

## print\_tensor

```
keras.backend.print_tensor(x, message='')
```

在评估时打印 `message` 和张量的值。

请注意，`print_tensor` 返回一个与 `x` 相同的新张量，应该在后面的代码中使用它。否则在评估过程中不会考虑打印操作。

## 示例

```
>>> x = K.print_tensor(x, message="x is: ")
```

## 参数

- **x**: 需要打印的张量。
- **message**: 需要与张量一起打印的消息。

## 返回

同一个不变的张量 `x`。

## function

```
keras.backend.function(inputs, outputs, updates=None)
```

实例化 Keras 函数。

## gradients

```
keras.backend.gradients(loss, variables)
```

返回 `variables` 在 `loss` 上的梯度。

## 参数

- **loss**: 需要最小化的标量张量。
- **variables**: 变量列表。

## 返回

一个梯度张量。

## stop\_gradient

```
keras.backend.stop_gradient(variables)
```

返回 `variables`，但是对于其他变量，其梯度为零。

### 参数

- **variables**: 需要考虑的张量或张量列表，任何的其他变量保持不变。

### 返回

单个张量或张量列表（取决于传递的参数），与任何其他变量具有恒定的梯度。

## rnn

```
keras.backend.rnn(step_function, inputs, initial_states, go_backwards=False,  
mask=None, constants=None, unroll=False, input_length=None)
```

在张量的时间维度迭代。

### 参数

- **step\_function**: RNN 步骤函数，
- **inputs**: 尺寸为 `(samples, ...)` 的张量 (不含时间维度)，表示批次样品在某个时间步的输入。
- **states**: 张量列表。
- **outputs**: 尺寸为 `(samples, output_dim)` 的张量 (不含时间维度)
- **new\_states**: 张量列表，与 `states` 长度和尺寸相同。列表中的第一个状态必须是前一个时间步的输出张量。
- **inputs**: 时序数据张量 `(samples, time, ...)` (最少 3D)。
- **initial\_states**: 尺寸为 `(samples, output_dim)` 的张量 (不含时间维度)，包含步骤函数中使用的状态的初始值。
- **go\_backwards**: 布尔值。如果为 `True`，以相反的顺序在时间维上进行迭代并返回相反的序列。
- **mask**: 尺寸为 `(samples, time, 1)` 的二进制张量，对于被屏蔽的每个元素都为零。
- **constants**: 每个步骤传递的常量值列表。
- **unroll**: 是否展开 RNN 或使用符号循环（依赖于后端的 `while_loop` 或 `scan`）。
- **input\_length**: 与 TensorFlow 实现不相关。如果使用 Theano 展开，则必须指定。

## 返回

一个元组，`(last_output, outputs, new_states)`。

- **last\_output**: rnn 的最后输出，尺寸为`(samples, ...)`。
- **outputs**: 尺寸为`(samples, time, ...)`的张量，其中每一项`outputs[s, t]`是样本 s 在时间 t 的步骤函数输出值。
- **new\_states**: 张量列表，有步骤函数返回的最后状态，尺寸为`(samples, ...)`。

## 异常

- **ValueError**: 如果输入的维度小于 3。
- **ValueError**: 如果`unroll` 为`True` 但输入时间步并不是固定的数字。
- **ValueError**: 如果提供了`mask` (非`None`) 但未提供`states` (`len(states) == 0`)。

## Numpy 实现

## 展示 Numpy 实现

```
def rnn(step_function, inputs, initial_states,
       go_backwards=False, mask=None, constants=None,
       unroll=False, input_length=None):
    if constants is None:
        constants = []
    output_sample, _ = step_function(inputs[:, 0], initial_states + constants)
    if mask is not None:
        if mask.dtype != np.bool:
            mask = mask.astype(np.bool)
        if mask.shape != inputs.shape[:2]:
            raise ValueError(
                'mask should have `shape=(samples, time)`', '
                'got {}'.format(mask.shape))
    def expand_mask(mask_, x):
        # expand mask so that `mask[:, t].ndim == x.ndim`
        while mask_.ndim < x.ndim + 1:
            mask_ = np.expand_dims(mask_, axis=-1)
        return mask_
    output_mask = expand_mask(mask, output_sample)
    states_masks = [expand_mask(mask, state) for state in initial_states]
    if input_length is None:
        input_length = inputs.shape[1]
    assert input_length == inputs.shape[1]
    time_index = range(input_length)
    if go_backwards:
        time_index = time_index[::-1]
    outputs = []
    states_tm1 = initial_states # tm1 means "t minus one" as in "previous
    timestep"
    output_tm1 = np.zeros(output_sample.shape)
    for t in time_index:
        output_t, states_t = step_function(inputs[:, t], states_tm1 +
constants)
        if mask is not None:
            output_t = np.where(output_mask[:, t], output_t, output_tm1)
            states_t = [np.where(state_mask[:, t], state_t, state_tm1)
                        for state_mask, state_t, state_tm1
                        in zip(states_masks, states_t, states_tm1)]
        outputs.append(output_t)
        states_tm1 = states_t
        output_tm1 = output_t
    return outputs[-1], np.stack(outputs, axis=1), states_tm1
```

## switch

```
keras.backend.switch(condition, then_expression, else_expression)
```

根据一个标量值在两个操作之间切换。

请注意，`then_expression` 和 `else_expression` 都应该是\*相同尺寸\*的符号张量。

## 参数

- **condition**: 张量 (`int` 或 `bool`)。
- **then\_expression**: 张量或返回张量的可调用函数。
- **else\_expression**: 张量或返回张量的可调用函数。

## 返回

选择的张量。

## 异常

- **ValueError**: 如果 `condition` 的秩大于两个表达式的秩。

## Numpy 实现

```
def switch(condition, then_expression, else_expression):  
    cond_float = condition.astype(floatx())  
    while cond_float.ndim < then_expression.ndim:  
        cond_float = cond_float[..., np.newaxis]  
    return cond_float * then_expression + (1 - cond_float) * else_expression
```

## in\_train\_phase

```
keras.backend.in_train_phase(x, alt, training=None)
```

在训练阶段选择 `x`，其他阶段选择 `alt`。

请注意 `alt` 应该与 `x` 尺寸相同。

## 参数

- **x**: 在训练阶段需要返回的 `x` (张量或返回张量的可调用函数)。
- **alt**: 在其他阶段需要返回的 `alt` (张量或返回张量的可调用函数)。
- **training**: 可选的标量张量 (或 Python 布尔值，或者 Python 整数)，以指定学习阶段。

## 返回

基于 `training` 标志，要么返回 `x`，要么返回 `alt`。`training` 标志默认为 `K.learning_phase()`。

## in\_test\_phase

```
keras.backend.in_test_phase(x, alt, training=None)
```

在测试阶段选择 `x`，其他阶段选择 `alt`。

请注意 `alt` 应该与 `x` 尺寸相同。

## 参数

- `x`: 在训练阶段需要返回的 `x` (张量或返回张量的可调用函数)。
- `alt`: 在其他阶段需要返回的 `alt` (张量或返回张量的可调用函数)。
- `training`: 可选的标量张量 (或 Python 布尔值, 或者 Python 整数)，以指定学习阶段。

## 返回

基于 `K.learning_phase`，要么返回 `x`，要么返回 `alt`。

## relu

```
keras.backend.relu(x, alpha=0.0, max_value=None)
```

ReLU 整流线性单位。

默认情况下，它返回逐个元素的 `max(x, 0)` 值。

## 参数

- `x`: 一个张量或变量。
- `alpha`: 一个标量，负数部分的斜率 (默认为 `0.`)。
- `max_value`: 饱和度阈值。

## 返回

一个张量。

## Numpy 实现

```
def relu(x, alpha=0., max_value=None, threshold=0.):  
    if max_value is None:  
        max_value = np.inf  
    above_threshold = x * (x >= threshold)  
    above_threshold = np.clip(above_threshold, 0.0, max_value)  
    below_threshold = alpha * (x - threshold) * (x < threshold)  
    return below_threshold + above_threshold
```

## elu

```
keras.backend.elu(x, alpha=1.0)
```

指数线性单元。

### 参数

- **x**: 用于计算激活函数的张量或变量。
- **alpha**: 一个标量，负数部分的斜率。

### 返回

一个张量。

### Numpy 实现

```
def elu(x, alpha=1.):
    return x * (x > 0) + alpha * (np.exp(x) - 1.) * (x < 0)
```

## softmax

```
keras.backend.softmax(x)
```

张量的 Softmax 值。

### 参数

- **x**: 张量或变量。

### 返回

一个张量。

### Numpy 实现

```
def softmax(x, axis=-1):
    y = np.exp(x - np.max(x, axis, keepdims=True))
    return y / np.sum(y, axis, keepdims=True)
```

## softplus

```
keras.backend.softplus(x)
```

张量的 Softplus 值。

## 参数

- **x**: 张量或变量。

## 返回

一个张量。

## Numpy 实现

```
def softplus(x):
    return np.log(1. + np.exp(x))
```

## softsign

```
keras.backend.softsign(x)
```

张量的 Softsign 值。

## 参数

- **x**: 张量或变量。

## 返回

一个张量。

## Numpy 实现

```
def softsign(x):
    return x / (1 + np.abs(x))
```

## categorical\_crossentropy

```
keras.backend.categorical_crossentropy(target, output, from_logits=False)
```

输出张量与目标张量之间的分类交叉熵。

## 参数

- **target**: 与 `output` 尺寸相同的张量。

- **output**: 由 softmax 产生的张量 (除非 `from_logits` 为 True, 在这种情况下 `output` 应该是对数形式)。
- **from\_logits**: 布尔值, `output` 是 softmax 的结果, 还是对数形式的张量。

## 返回

输出张量。

## sparse\_categorical\_crossentropy

```
keras.backend.sparse_categorical_crossentropy(target, output,  
from_logits=False)
```

稀疏表示的整数值目标的分类交叉熵。

## 参数

- **target**: 一个整数张量。
- **output**: 由 softmax 产生的张量 (除非 `from_logits` 为 True, 在这种情况下 `output` 应该是对数形式)。
- **from\_logits**: 布尔值, `output` 是 softmax 的结果, 还是对数形式的张量。

## 返回

输出张量。

## binary\_crossentropy

```
keras.backend.binary_crossentropy(target, output, from_logits=False)
```

输出张量与目标张量之间的二进制交叉熵。

## 参数

- **target**: 与 `output` 尺寸相同的张量。
- **output**: 一个张量。
- **from\_logits**: `output` 是否是对数张量。默认情况下, 我们认为 `output` 编码了概率分布。

## 返回

一个张量。

## sigmoid

```
keras.backend.sigmoid(x)
```

逐个元素求 sigmoid 值。

### 参数

- **x**: 一个张量或变量。

### 返回

一个张量。

### Numpy 实现

```
def sigmoid(x):  
    return 1. / (1. + np.exp(-x))
```

## hard\_sigmoid

```
keras.backend.hard_sigmoid(x)
```

分段的 sigmoid 线性近似。速度比 sigmoid 更快。

- 如果  $x < -2.5$ ，返回 0。
- 如果  $x > 2.5$ ，返回 1。
- 如果  $-2.5 \leq x \leq 2.5$ ，返回  $0.2 * x + 0.5$ 。

### 参数

- **x**: 一个张量或变量。

### 返回

一个张量。

### Numpy 实现

```
def hard_sigmoid(x):  
    y = 0.2 * x + 0.5  
    return np.clip(y, 0, 1)
```

## tanh

```
keras.backend.tanh(x)
```

逐个元素求 tanh 值。

### 参数

- **x**: 一个张量或变量。

### 返回

一个张量。

## Numpy 实现

```
def tanh(x):  
    return np.tanh(x)
```

## dropout

```
keras.backend.dropout(x, level, noise_shape=None, seed=None)
```

将 **x** 中的某些项随机设置为零，同时缩放整个张量。

### 参数

- **x**: 张量
- **level**: 张量中将被设置为 0 的项的比例。
- **noise\_shape**: 随机生成的 保留/丢弃 标志的尺寸， 必须可以广播到 **x** 的尺寸。
- **seed**: 保证确定性的随机种子。

### 返回

一个张量。

## Numpy 实现

## 展示 Numpy 实现

```
def dropout(x, level, noise_shape=None, seed=None):
    if noise_shape is None:
        noise_shape = x.shape
    if learning_phase():
        noise = np.random.choice([0, 1],
                               noise_shape,
                               replace=True,
                               p=[level, 1 - level])
        return x * noise / (1 - level)
    else:
        return x
```

## l2\_normalize

```
keras.backend.l2_normalize(x, axis=None)
```

在指定的轴使用 L2 范式 标准化一个张量。

### 参数

- **x**: 张量或变量。
- **axis**: 需要执行标准化的轴。

### 返回

一个张量。

## Numpy 实现

```
def l2_normalize(x, axis=-1):
    y = np.max(np.sum(x ** 2, axis, keepdims=True), axis, keepdims=True)
    return x / np.sqrt(y)
```

## in\_top\_k

```
keras.backend.in_top_k(predictions, targets, k)
```

判断 `targets` 是否在 `predictions` 的前 `k` 个中。

### 参数

- **predictions**: 一个张量，尺寸为 `(batch_size, classes)`，类型为 `float32`。

- **targets**: 一个 1D 张量，长度为 `batch_size`，类型为 `int32` 或 `int64`。
- **k**: 一个 `int`，要考虑的顶部元素的数量。

## 返回

一个 1D 张量，长度为 `batch_size`，类型为 `bool`。如果 `predictions[i, targets[i]]` 在 `predictions[i]` 的 top-`k` 值中，则 `output[i]` 为 `True`。

## conv1d

```
keras.backend.conv1d(x, kernel, strides=1, padding='valid', data_format=None,  
dilation_rate=1)
```

1D 卷积。

## 参数

- **x**: 张量或变量。
- **kernel**: 核张量。
- **strides**: 步长整型。
- **padding**: 字符串，`"same"`, `"causal"` 或 `"valid"`。
- **data\_format**: 字符串，`"channels_last"` 或 `"channels_first"`。
- **dilation\_rate**: 整数膨胀率。

## 返回

一个张量，1D 卷积结果。

## 异常

- **ValueError**: 如果 `data_format` 既不是 `channels_last` 也不是 `channels_first`。

## conv2d

```
keras.backend.conv2d(x, kernel, strides=(1, 1), padding='valid',  
data_format=None, dilation_rate=(1, 1))
```

2D 卷积。

## 参数

- **x**: 张量或变量。
- **kernel**: 核张量。
- **strides**: 步长元组。
- **padding**: 字符串, `"same"` 或 `"valid"`。
- **data\_format**: 字符串, `"channels_last"` 或 `"channels_first"`。对于输入/卷积核/输出, 是否使用 Theano 或 TensorFlow/CNTK数据格式。
- **dilation\_rate**: 2 个整数的元组。

## 返回

一个张量, 2D 卷积结果。

## 异常

- **ValueError**: 如果 `data_format` 既不是 `channels_last` 也不是 `channels_first`。

## conv2d\_transpose

```
keras.backend.conv2d_transpose(x, kernel, output_shape, strides=(1, 1),
padding='valid', data_format=None)
```

2D 反卷积 (即转置卷积)。

## 参数

- **x**: 张量或变量。
- **kernel**: 核张量。
- **output\_shape**: 表示输出尺寸的 1D 整型张量。
- **strides**: 步长元组。
- **padding**: 字符串, `"same"` 或 `"valid"`。
- **data\_format**: 字符串, `"channels_last"` 或 `"channels_first"`。对于输入/卷积核/输出, 是否使用 Theano 或 TensorFlow/CNTK数据格式。

## 返回

一个张量, 转置的 2D 卷积的结果。

## 异常

- **ValueError**: 如果 `data_format` 既不是 `channels_last` 也不是 `channels_first`。

## separable\_conv1d

```
keras.backend.separable_conv1d(x, depthwise_kernel, pointwise_kernel,  
strides=1, padding='valid', data_format=None, dilation_rate=1)
```

带可分离滤波器的 1D 卷积。

## 参数

- **x**: 输入张量。
- **depthwise\_kernel**: 用于深度卷积的卷积核。
- **pointwise\_kernel**: 1x1 卷积核。
- **strides**: 步长整数。
- **padding**: 字符串，`"same"` 或 `"valid"`。
- **data\_format**: 字符串，`"channels_last"` 或 `"channels_first"`。
- **dilation\_rate**: 整数膨胀率。

## 返回

输出张量。

## 异常

- **ValueError**: 如果 `data_format` 既不是 `channels_last` 也不是 `channels_first`。

## separable\_conv2d

```
keras.backend.separable_conv2d(x, depthwise_kernel, pointwise_kernel,  
strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1))
```

带可分离滤波器的 2D 卷积。

## 参数

- **x**: 输入张量。
- **depthwise\_kernel**: 用于深度卷积的卷积核。

- **pointwise\_kernel**: 1x1 卷积核。
- **strides**: 步长元组 (长度为 2)。
- **padding**: 字符串, "same" 或 "valid"。
- **data\_format**: 字符串, "channels\_last" 或 "channels\_first"。
- **dilation\_rate**: 整数元组, 可分离卷积的膨胀率。

## 返回

输出张量。

## 异常

- **ValueError**: 如果 `data_format` 既不是 `channels_last` 也不是 `channels_first`。

## depthwise\_conv2d

```
keras.backend.depthwise_conv2d(x, depthwise_kernel, strides=(1, 1),  
padding='valid', data_format=None, dilation_rate=(1, 1))
```

带可分离滤波器的 2D 卷积。

## 参数

- **x**: 输入张量。
- **depthwise\_kernel**: 用于深度卷积的卷积核。
- **strides**: 步长元组 (长度为 2)。
- **padding**: 字符串, "same" 或 "valid"。
- **data\_format**: 字符串, "channels\_last" 或 "channels\_first"。
- **dilation\_rate**: 整数元组, 可分离卷积的膨胀率。

## 返回

输出张量。

## 异常

- **ValueError**: 如果 `data_format` 既不是 `channels_last` 也不是 `channels_first`。

## conv3d

```
keras.backend.conv3d(x, kernel, strides=(1, 1, 1), padding='valid',  
data_format=None, dilation_rate=(1, 1, 1))
```

3D 卷积。

## 参数

- **x**: 张量或变量。
- **kernel**: 核张量。
- **strides**: 步长元组。
- **padding**: 字符串, "same" 或 "valid"。
- **data\_format**: 字符串, "channels\_last" 或 "channels\_first"。
- **dilation\_rate**: 3 个整数的元组。

## 返回

一个张量，3D 卷积的结果。

## 异常

- **ValueError**: 如果 `data_format` 既不是 `channels_last` 也不是 `channels_first`。

## conv3d\_transpose

```
keras.backend.conv3d_transpose(x, kernel, output_shape, strides=(1, 1, 1),  
padding='valid', data_format=None)
```

3D 反卷积 (即转置卷积)。

## 参数

- **x**: 输入张量。
- **kernel**: 核张量。
- **output\_shape**: 表示输出尺寸的 1D 整数张量。
- **strides**: 步长元组。
- **padding**: 字符串, "same" 或 "valid"。
- **data\_format**: 字符串, "channels\_last" 或 "channels\_first"。对于输入/卷积核/输出，是否使用 Theano 或 TensorFlow/CNTK 数据格式。

## 返回

一个张量，3D 转置卷积的结果。

## 异常

- **ValueError**: 如果 `data_format` 既不是 `channels_last` 也不是 `channels_first`。

## pool2d

```
keras.backend.pool2d(x, pool_size, strides=(1, 1), padding='valid',
                      data_format=None, pool_mode='max')
```

2D 池化。

## 参数

- **x**: 张量或变量。
- **pool\_size**: 2 个整数的元组。
- **strides**: 2 个整数的元组。
- **padding**: 字符串，`"same"` 或 `"valid"`。
- **data\_format**: 字符串，`"channels_last"` 或 `"channels_first"`。
- **pool\_mode**: 字符串，`"max"` 或 `"avg"`。

## 返回

一个张量，2D 池化的结果。

## 异常

- **ValueError**: 如果 `data_format` 既不是 `channels_last` 也不是 `channels_first`。
- **ValueError**: if `pool_mode` 既不是 `"max"` 也不是 `"avg"`。

## pool3d

```
keras.backend.pool3d(x, pool_size, strides=(1, 1, 1), padding='valid',
                      data_format=None, pool_mode='max')
```

3D 池化。

## 参数

- **x**: 张量或变量。

- **pool\_size**: 3 个整数的元组。
- **strides**: 3 个整数的元组。
- **padding**: 字符串, "same" 或 "valid"。
- **data\_format**: 字符串, "channels\_last" 或 "channels\_first"。
- **pool\_mode**: 字符串, "max" 或 "avg"。

## 返回

一个张量，3D 池化的结果。

## 异常

- **ValueError**: 如果 `data_format` 既不是 `channels_last` 也不是 `channels_first`。
- **ValueError**: if `pool_mode` 既不是 "max" 也不是 "avg"。

## local\_conv1d

```
keras.backend.local_conv1d(inputs, kernel, kernel_size, strides,
                           data_format=None)
```

在不共享权值的情况下，运用 1D 卷积。

## 参数

- **inputs**: 3D 张量，尺寸为 (`batch_size, steps, input_dim`)
- **kernel**: 卷积的非共享权重，尺寸为 (`output_items, feature_dim, filters`)
- **kernel\_size**: 一个整数的元组，指定 1D 卷积窗口的长度。
- **strides**: 一个整数的元组，指定卷积步长。
- **data\_format**: 数据格式，`channels_first` 或 `channels_last`。

## 返回

运用不共享权重的 1D 卷积之后的张量，尺寸为 (`batch_size, output_length, filters`)。

## 异常

- **ValueError**: 如果 `data_format` 既不是 `channels_last` 也不是 `channels_first`。

## local\_conv2d

```
keras.backend.local_conv2d(inputs, kernel, kernel_size, strides, output_shape,  
data_format=None)
```

在不共享权值的情况下，运用 2D 卷积。

## 参数

- **inputs**: 如果 `data_format='channels_first'`， 则为尺寸为  $(batch\_size, filters, new\_rows, new\_cols)$  的 4D 张量。如果 `data_format='channels_last'`， 则为尺寸为  $(batch\_size, new\_rows, new\_cols, filters)$  的 4D 张量。
- **kernel**: 卷积的非共享权重, 尺寸为  $(output\_items, feature\_dim, filters)$
- **kernel\_size**: 2 个整数的元组， 指定 2D 卷积窗口的宽度和高度。
- **strides**: 2 个整数的元组， 指定 2D 卷积沿宽度和高度方向的步长。
- **output\_shape**: 元组  $(output\_row, output\_col)$ 。
- **data\_format**: 数据格式， `channels_first` 或 `channels_last`。

## 返回

一个 4D 张量。

- 如果 `data_format='channels_first'`， 尺寸为  $(batch\_size, filters, new\_rows, new\_cols)$ 。
- 如果 `data_format='channels_last'`， 尺寸为  $(batch\_size, new\_rows, new\_cols, filters)$

## 异常

- **ValueError**: 如果 `data_format` 既不是 `channels_last` 也不是 `channels_first`。

## bias\_add

```
keras.backend.bias_add(x, bias, data_format=None)
```

给张量添加一个偏置向量。

## 参数

- **x**: 张量或变量。
- **bias**: 需要添加的偏置向量。
- **data\_format**: 字符串， `"channels_last"` 或 `"channels_first"`。

## 返回

输出张量。

## 异常

- **ValueError**: 以下两种情况之一：
  - 无效的 `data_format` 参数。
  - 无效的偏置向量尺寸。偏置应该是一个 `ndim(x)-1` 维的向量或张量。

## Numpy 实现

### 展示 Numpy 实现

```
def bias_add(x, y, data_format):
    if data_format == 'channels_first':
        if y.ndim > 1:
            y = np.reshape(y, y.shape[::-1])
        for _ in range(x.ndim - y.ndim - 1):
            y = np.expand_dims(y, -1)
    else:
        for _ in range(x.ndim - y.ndim - 1):
            y = np.expand_dims(y, 0)
    return x + y
```

## random\_normal

```
keras.backend.random_normal(shape, mean=0.0, stddev=1.0, dtype=None,
seed=None)
```

返回正态分布值的张量。

### 参数

- **shape**: 一个整数元组，需要创建的张量的尺寸。
- **mean**: 一个浮点数，抽样的正态分布平均值。
- **stddev**: 一个浮点数，抽样的正态分布标准差。
- **dtype**: 字符串，返回的张量的数据类型。
- **seed**: 整数，随机种子。

### 返回

一个张量。

## random\_uniform

```
keras.backend.random_uniform(shape, minval=0.0, maxval=1.0, dtype=None,  
seed=None)
```

返回均匀分布值的张量。

## 参数

- **shape**: 一个整数元组，需要创建的张量的尺寸。
- **minval**: 一个浮点数，抽样的均匀分布下界。
- **maxval**: 一个浮点数，抽样的均匀分布上界。
- **dtype**: 字符串，返回的张量的数据类型。
- **seed**: 整数，随机种子。

## 返回

一个张量。

## random\_binomial

```
keras.backend.random_binomial(shape, p=0.0, dtype=None, seed=None)
```

返回随机二项分布值的张量。

## 参数

- **shape**: 一个整数元组，需要创建的张量的尺寸。
- **p**: 一个浮点数， $0. \leq p \leq 1$ ，二项分布的概率。
- **dtype**: 字符串，返回的张量的数据类型。
- **seed**: 整数，随机种子。

## 返回

一个张量。

## truncated\_normal

```
keras.backend.truncated_normal(shape, mean=0.0, stddev=1.0, dtype=None,  
seed=None)
```

返回截断的随机正态分布值的张量。

生成的值遵循具有指定平均值和标准差的正态分布，此外，其中数值大于平均值两个标准差的将被丢弃和重新挑选。

## 参数

- **shape**: 一个整数元组，需要创建的张量的尺寸。
- **mean**: 平均值。
- **stddev**: 标准差。
- **dtype**: 字符串，返回的张量的数据类型。
- **seed**: 整数，随机种子。

## 返回

一个张量。

## ctc\_label\_dense\_to\_sparse

```
keras.backend.ctc_label_dense_to_sparse(labels, label_lengths)
```

将 CTC 标签从密集转换为稀疏表示。

## 参数

- **labels**: 密集 CTC 标签。
- **label\_lengths**: 标签长度。

## 返回

一个表示标签的稀疏张量。

## ctc\_batch\_cost

```
keras.backend.ctc_batch_cost(y_true, y_pred, input_length, label_length)
```

在每个批次元素上运行 CTC 损失算法。

## 参数

- **y\_true**: 张量 `(samples, max_string_length)`，包含真实标签。
- **y\_pred**: 张量 `(samples, time_steps, num_categories)`，包含预测值，或 softmax 输出。

- **input\_length**: 张量 `(samples, 1)`，包含 `y_pred` 中每个批次样本的序列长度。
- **label\_length**: 张量 `(samples, 1)`，包含 `y_true` 中每个批次样本的序列长度。

## 返回

尺寸为 `(samples,1)` 的张量，包含每一个元素的 CTC 损失。

## ctc\_decode

```
keras.backend.ctc_decode(y_pred, input_length, greedy=True, beam_width=100,  
top_paths=1, merge_repeated=False)
```

解码 softmax 的输出。

可以使用贪心搜索（也称为最优路径）或受限字典搜索。

## 参数

- **y\_pred**: 张量 `(samples, time_steps, num_categories)`，包含预测值，或 softmax 输出。
- **input\_length**: 张量 `(samples,)`，包含 `y_pred` 中每个批次样本的序列长度。
- **greedy**: 如果为 `True`，则执行更快速的最优路径搜索，而不使用字典。
- **beam\_width**: 如果 `greedy` 为 `False`，将使用该宽度的 beam 搜索解码器搜索。
- **top\_paths**: 如果 `greedy` 为 `else`，将返回多少条最可能的路径。

## 返回

- **Tuple**:
- **List**: 如果 `greedy` 为 `True`，返回包含解码序列的一个元素的列表。如果为 `False`，返回最可能解码序列的 `top_paths`。
- **Important**: 空白标签返回为 `-1`。包含每个解码序列的对数概率的张量 `(top_paths, )`。

## control\_dependencies

```
keras.backend.control_dependencies(control_inputs)
```

一个指定控制依赖的上下文管理器。

## 参数

- **control\_inputs**: 一系列的对象的操作或张量，它们必须在执行上下文中定义的操作之前执行。它也可以是 `None`，表示清空控制依赖。

## Returns

一个上下文管理器。

## map\_fn

```
keras.backend.map_fn(fn, elems, name=None, dtype=None)
```

将函数fn映射到元素 `elems` 上并返回输出。

## 参数

- **fn**: 将在每个元素上调用的可调用函数。
- **elems**: 张量。
- **name**: 映射节点在图中的字符串名称。
- **dtype**: 输出数据格式。

## 返回

数据类型为 `dtype` 的张量。

## foldl

```
keras.backend.foldl(fn, elems, initializer=None, name=None)
```

使用 fn 归约 `elems`，以从左到右组合它们。

## 参数

- **fn**: 将在每个元素和一个累加器上调用的可调用函数，例如 `lambda acc, x: acc + x`。
- **elems**: 张量。
- **initializer**: 第一个使用的值 (如果为 `None`，使用 `elems[0]` )。
- **name**: `foldl` 节点在图中的字符串名称。

## 返回

与 `initializer` 类型和尺寸相同的张量。

## foldr

```
keras.backend.foldr(fn, elems, initializer=None, name=None)
```

使用 `fn` 归约 `elems`，以从右到左组合它们。

### 参数

- **fn**: 将在每个元素和一个累加器上调用的可调用函数，例如 `lambda acc, x: acc + x`。
- **elems**: 张量。
- **initializer**: 第一个使用的值 (如果为 `None`，使用 `elems[-1]`)。
- **name**: `foldr` 节点在图中的字符串名称。

### 返回

与 `initializer` 类型和尺寸相同的张量。



# 初始化 Initializers

## 初始化器的用法

初始化定义了设置 Keras 各层权重随机初始值的方法。

用来将初始化器传入 Keras 层的参数名取决于具体的层。通常关键字为 `kernel_initializer` 和 `bias_initializer`：

```
model.add(Dense(64,  
               kernel_initializer='random_uniform',  
               bias_initializer='zeros'))
```

## 预定义初始化器

下面这些是可用的内置初始化器，是 `keras.initializers` 模块的一部分：

**Initializer**

```
keras.initializers.Initializer()
```

初始化器基类：所有初始化器继承这个类。

Zeros

```
keras.initializers.Zeros()
```

将张量初始值设为 0 的初始化器。

Ones

```
keras.initializers.Ones()
```

将张量初始值设为 1 的初始化器。

Constant

```
keras.initializers.Constant(value=0)
```

将张量初始值设为一个常数的初始化器。

## 参数

- **value**: 浮点数，生成的张量的值。

RandomNormal

```
keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed=None)
```

按照正态分布生成随机张量的初始化器。

## 参数

- **mean**: 一个 Python 标量或者一个标量张量。要生成的随机值的平均数。
- **stddev**: 一个 Python 标量或者一个标量张量。要生成的随机值的标准差。
- **seed**: 一个 Python 整数。用于设置随机数种子。

RandomUniform

```
keras.initializers.RandomUniform(minval=-0.05, maxval=0.05, seed=None)
```

按照均匀分布生成随机张量的初始化器。

## 参数

- **minval**: 一个 Python 标量或者一个标量张量。要生成的随机值的范围下限。
- **maxval**: 一个 Python 标量或者一个标量张量。要生成的随机值的范围下限。默认为浮点类型的 1。
- **seed**: 一个 Python 整数。用于设置随机数种子。

TruncatedNormal

```
keras.initializers.TruncatedNormal(mean=0.0, stddev=0.05, seed=None)
```

按照截尾正态分布生成随机张量的初始化器。

生成的随机值与 `RandomNormal` 生成的类似，但是在距离平均值两个标准差之外的随机值将被丢弃并重新生成。这是用来生成神经网络权重和滤波器的推荐初始化器。

### Arguments

- **mean**: 一个 Python 标量或者一个标量张量。要生成的随机值的平均数。
- **stddev**: 一个 Python 标量或者一个标量张量。要生成的随机值的标准差。
- **seed**: 一个 Python 整数。用于设置随机数种子。

VarianceScaling

```
keras.initializers.VarianceScaling(scale=1.0, mode='fan_in',
distribution='normal', seed=None)
```

初始化器能够根据权值的尺寸调整其规模。

使用 `distribution="normal"` 时，样本是从一个以 0 为中心的截断正态分布中抽取的，  
`stddev = sqrt(scale / n)`，其中 `n` 是：

- 权值张量中输入单元的数量，如果 `mode = "fan_in"`。
- 输出单元的数量，如果 `mode = "fan_out"`。
- 输入和输出单位数量的平均数，如果 `mode = "fan_avg"`。

使用 `distribution="uniform"` 时，样本是从  $[-\text{limit}, \text{limit}]$  内的均匀分布中抽取的，其中  
`limit = sqrt(3 * scale / n)`。

## 参数

- **scale**: 缩放因子（正浮点数）。
- **mode**: “fan\_in”, “fan\_out”, “fan\_avg” 之一。
- **distribution**: 使用的随机分布。“normal”, “uniform” 之一。
- **seed**: 一个 Python 整数。作为随机发生器的种子。

## 异常

- **ValueError**: 如果 “scale”, `mode` 或 “distribution” 参数无效。

## Orthogonal

```
keras.initializers.Orthogonal(gain=1.0, seed=None)
```

生成一个随机正交矩阵的初始化器。

## 参数

- **gain**: 适用于正交矩阵的乘法因子。
- **seed**: 一个 Python 整数。作为随机发生器的种子。

## 参考文献

- [Exact solutions to the nonlinear dynamics of learning in deep linear neural networks](#)

## Identity

[\[source\]](#)

```
keras.initializers.Identity(gain=1.0)
```

生成单位矩阵的初始化器。

仅用于 2D 方阵。

## 参数

- **gain**: 适用于单位矩阵的乘法因子。

## lecun\_uniform

```
keras.initializers.lecun_uniform(seed=None)
```

LeCun 均匀初始化器。

它从  $[-\text{limit}, \text{limit}]$  中的均匀分布中抽取样本，其中 `limit` 是  $\sqrt{3 / \text{fan\_in}}$ ，`fan_in` 是权值张量中的输入单位的数量。

## 参数

- **seed**: 一个 Python 整数。作为随机发生器的种子。

## 返回

一个初始化器。

## 参考文献

- [Efficient Backprop](#)

## glorot\_normal

```
keras.initializers.glorot_normal(seed=None)
```

Glorot 正态分布初始化器，也称为 Xavier 正态分布初始化器。

它从以 0 为中心，标准差为 `stddev = sqrt(2 / (fan_in + fan_out))` 的截断正态分布中抽取样本，其中 `fan_in` 是权值张量中的输入单位的数量，`fan_out` 是权值张量中的输出单位的数量。

### 参数

- `seed`: 一个 Python 整数。作为随机发生器的种子。

### 返回

一个初始化器。

### 参考文献

- [Understanding the difficulty of training deep feedforward neural networks](#)

## glorot\_uniform

```
keras.initializers.glorot_uniform(seed=None)
```

Glorot 均匀分布初始化器，也称为 Xavier 均匀分布初始化器。

它从  $[-\text{limit}, \text{limit}]$  中的均匀分布中抽取样本，其中 `limit` 是  $\sqrt{6 / (\text{fan\_in} + \text{fan\_out})}$ ，`fan_in` 是权值张量中的输入单位的数量，`fan_out` 是权值张量中的输出单位的数量。

### 参数

- `seed`: 一个 Python 整数。作为随机发生器的种子。

### 返回

一个初始化器。

### 参考文献

- [Understanding the difficulty of training deep feedforward neural networks](#)

## he\_normal

```
keras.initializers.he_normal(seed=None)
```

He 正态分布初始化器。

它从以 0 为中心，标准差为 `stddev = sqrt(2 / fan_in)` 的截断正态分布中抽取样本，其中 `fan_in` 是权值张量中的输入单位的数量，

## 参数

- **seed**: 一个 Python 整数。作为随机发生器的种子。

## 返回

一个初始化器。

## 参考文献

- [Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification](#)

## lecun\_normal

```
keras.initializers.lecun_normal(seed=None)
```

LeCun 正态分布初始化器。

它从以 0 为中心，标准差为 `stddev = sqrt(1 / fan_in)` 的截断正态分布中抽取样本，其中 `fan_in` 是权值张量中的输入单位的数量。

## 参数

- **seed**: 一个 Python 整数。作为随机发生器的种子。

## 返回

一个初始化器。

## 参考文献

- [Self-Normalizing Neural Networks](#)
- [Efficient Backprop](#)

## he\_uniform

```
keras.initializers.he_uniform(seed=None)
```

He 均匀方差缩放初始化器。

它从  $[-\text{limit}, \text{limit}]$  中的均匀分布中抽取样本，其中 `limit` 是  $\sqrt{6 / \text{fan\_in}}$ ，其中 `fan_in` 是权值张量中的输入单位的数量。

## 参数

- `seed`: 一个 Python 整数。作为随机发生器的种子。

## 返回

一个初始化器。

## 参考文献

- [Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification](#)

一个初始化器可以作为一个字符串传递（必须匹配上面的一个可用的初始化器），或者作为一个可调用函数传递：

```
from keras import initializers

model.add(Dense(64,
    kernel_initializer=initializers.random_normal(stddev=0.01)))

# 同样有效；将使用默认参数。
model.add(Dense(64, kernel_initializer='random_normal'))
```

## 使用自定义初始化器

如果传递一个自定义的可调用函数，那么它必须使用参数 `shape`（需要初始化的变量的尺寸）和 `dtype`（数据类型）：

```
from keras import backend as K

def my_init(shape, dtype=None):
    return K.random_normal(shape, dtype=dtype)

model.add(Dense(64, kernel_initializer=my_init))
```



# 正则化 Regularizers

## 正则化器的使用

正则化器允许在优化过程中对层的参数或层的激活情况进行惩罚。网络优化的损失函数也包括这些惩罚项。

惩罚是以层为对象进行的。具体的 API 因层而异，但 `Dense`，`Conv1D`，`Conv2D` 和 `Conv3D` 这些层具有统一的 API。

正则化器开放 3 个关键字参数：

- `kernel_regularizer`: `keras.regularizers.Regularizer` 的实例
- `bias_regularizer`: `keras.regularizers.Regularizer` 的实例
- `activity_regularizer`: `keras.regularizers.Regularizer` 的实例

## 示例

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
               kernel_regularizer=regularizers.l2(0.01),
               activity_regularizer=regularizers.l1(0.01)))
```

## 可用的正则化器

```
keras.regularizers.l1(0.)
keras.regularizers.l2(0.)
keras.regularizers.l1_l2(l1=0.01, l2=0.01)
```

## 开发新的正则化器

任何输入一个权重矩阵、返回一个损失贡献张量的函数，都可以用作正则化器，例如：

```
from keras import backend as K

def l1_reg(weight_matrix):
    return 0.01 * K.sum(K.abs(weight_matrix))

model.add(Dense(64, input_dim=64,
               kernel_regularizer=l1_reg))
```

另外，你也可以用面向对象的方式来编写正则化器的代码，例子见 [keras/regularizers.py](#) 模块。



# 约束项 Constraints

## 约束项的使用

`constraints` 模块的函数允许在优化期间对网络参数设置约束（例如非负性）。

约束是以层为对象进行的。具体的 API 因层而异，但 `Dense`，`Conv1D`，`Conv2D` 和 `Conv3D` 这些层具有统一的 API。

约束层开放 2 个关键字参数：

- `kernel_constraint` 用于主权重矩阵。
- `bias_constraint` 用于偏置。

```
from keras.constraints import max_norm
model.add(Dense(64, kernel_constraint=max_norm(2.)))
```

## 预定义的约束

MaxNorm

```
keras.constraints.MaxNorm(max_value=2, axis=0)
```

MaxNorm 最大范数权值约束。

映射到每个隐藏单元的权值的约束，使其具有小于或等于期望值的范数。

## 参数

- **max\_value**: 输入权值的最大范数。
- **axis**: 整数，需要计算权值范数的轴。例如，在 Dense 层中权值矩阵的尺寸为 `(input_dim, output_dim)`，设置 `axis` 为 `0` 以约束每个长度为 `(input_dim,)` 的权值向量。在 Conv2D 层 (`data_format="channels_last"`) 中，权值张量的尺寸为 `(rows, cols, input_depth, output_depth)`，设置 `axis` 为 `[0, 1, 2]` 以越是每个尺寸为 `(rows, cols, input_depth)` 的滤波器张量的权值。

## 参考文献

- [Dropout: A Simple Way to Prevent Neural Networks from Overfitting] (<http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf>)

## NonNeg

```
keras.constraints.NonNeg()
```

权重非负的约束。

## UnitNorm

```
keras.constraints.UnitNorm(axis=0)
```

映射到每个隐藏单元的权值的约束，使其具有单位范数。

### 参数

- **axis**: 整数，需要计算权值范数的轴。例如，在 Dense 层中权值矩阵的尺寸为 `(input_dim, output_dim)`，设置 `axis` 为 `0` 以约束每个长度为 `(input_dim,)` 的权值向量。在 Conv2D 层 (`data_format="channels_last"`) 中，权值张量的尺寸为 `(rows, cols, input_depth, output_depth)`，设置 `axis` 为 `[0, 1, 2]` 以越是每个尺寸为 `(rows, cols, input_depth)` 的滤波器张量的权值。

## MinMaxNorm

[\[source\]](#)

```
keras.constraints.MinMaxNorm(min_value=0.0, max_value=1.0, rate=1.0, axis=0)
```

MinMaxNorm 最小/最大范数权值约束。

映射到每个隐藏单元的权值的约束，使其范数在上下界之间。

### 参数

- **min\_value**: 输入权值的最小范数。

- **max\_value**: 输入权值的最大范数。
- **rate**: 强制执行约束的比例：权值将被重新调整为 `(1 - rate) * norm + rate * norm.clip(min_value, max_value)`。实际上，这意味着 `rate = 1.0` 代表严格执行约束，而 `rate < 1.0` 意味着权值将在每一步重新调整以缓慢移动到所需间隔内的值。
- **axis**: 整数，需要计算权值范数的轴。例如，在 `Dense` 层中权值矩阵的尺寸为 `(input_dim, output_dim)`，设置 `axis` 为 `0` 以约束每个长度为 `(input_dim,)` 的权值向量。在 `Conv2D` 层 (`data_format="channels_last"`) 中，权值张量的尺寸为 `(rows, cols, input_depth, output_depth)`，设置 `axis` 为 `[0, 1, 2]` 以越是每个尺寸为 `(rows, cols, input_depth)` 的滤波器张量的权值。



# 可视化 Visualization

## 模型可视化

`keras.utils.vis_utils` 模块提供了一些绘制 Keras 模型的实用功能(使用 `graphviz`)。

以下实例，将绘制一张模型图，并保存为文件：

```
from keras.utils import plot_model
plot_model(model, to_file='model.png')
```

`plot_model` 有 4 个可选参数：

- `show_shapes` (默认为 `False`) 控制是否在图中输出各层的尺寸。
- `show_layer_names` (默认为 `True`) 控制是否在图中显示每一层的名字。
- `expand_dim` (默认为 `False`) 控制是否将嵌套模型扩展为图形中的聚类。
- `dpi` (默认为 96) 控制图像 `dpi`。

此外，你也可以直接取得 `pydot.Graph` 对象并自己渲染它。例如，ipython notebook 中的可视化实例如下：

```
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot

SVG(model_to_dot(model).create(prog='dot', format='svg'))
```

## 训练历史可视化

Keras Model 上的 `fit()` 方法返回一个 `History` 对象。`History.history` 属性是一个记录了连续迭代的训练/验证（如果存在）损失值和评估值的字典。这里是一个简单的使用 `matplotlib` 来生成训练/验证集的损失和准确率图表的例子：

```
import matplotlib.pyplot as plt

history = model.fit(x, y, validation_split=0.25, epochs=50, batch_size=16,
verbose=1)

# 绘制训练 & 验证的准确率值
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
```

```
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

# 绘制训练 & 验证的损失值
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```



# Scikit-Learn API 的封装器

你可以使用 Keras 的 `Sequential` 模型（仅限单一输入）作为 Scikit-Learn 工作流程的一部分，通过在此找到的包装器: `keras.wrappers.scikit_learn.py`。

有两个封装器可用:

`keras.wrappers.scikit_learn.KerasClassifier(build_fn=None, **sk_params)`, 这实现了 Scikit-Learn 分类器接口,

`keras.wrappers.scikit_learn.KerasRegressor(build_fn=None, **sk_params)`, 这实现了 Scikit-Learn 回归接口。

## 参数

- `build_fn`: 可调用函数或类实例
- `sk_params`: 模型参数和拟合参数

`build_fn` 应该建立, 编译, 并返回一个 Keras 模型, 然后被用来训练/预测。以下三个值之一可以传递给 `build_fn`

1. 一个函数;
2. 实现 `__call__` 方法的类的实例;
3. `None`。这意味着你实现了一个继承自 `KerasClassifier` 或 `KerasRegressor` 的类。当前类 `__call__` 方法将被视为默认的 `build_fn`。

`sk_params` 同时包含模型参数和拟合参数。合法的模型参数是 `build_fn` 的参数。请注意, 与 scikit-learn 中的所有其他估算器一样, `build_fn` 应为其参数提供默认值, 以便你可以创建估算器而不将任何值传递给 `sk_params`。

`sk_params` 还可以接受用于调用 `fit`, `predict`, `predict_proba` 和 `score` 方法的参数 (例如, `epochs`, `batch_size`)。训练 (预测) 参数按以下顺序选择:

1. 传递给 `fit`, `predict`, `predict_proba` 和 `score` 函数的字典参数的值;
2. 传递给 `sk_params` 的值;
3. `keras.models.Sequential` 的 `fit`, `predict`, `predict_proba` 和 `score` 方法的默认值。

当使用 scikit-learn 的 `grid_search` API 时, 合法可调参数是你可以传递给 `sk_params` 的参数, 包括训练参数。换句话说, 你可以使用 `grid_search` 来搜索最佳的 `batch_size` 或 `epoch` 以及其他模型参数。



# 工具 Utils

CustomObjectScope

```
keras.utils.CustomObjectScope()
```

提供更改为 `_GLOBAL_CUSTOM_OBJECTS` 无法转义的范围。

`with` 语句中的代码将能够通过名称访问自定义对象。对全局自定义对象的更改会在封闭的 `with` 语句中持续存在。在 `with` 语句结束时，全局自定义对象将恢复到 `with` 语句开始时的状态。

## 示例

考虑自定义对象 `MyObject` (例如一个类)：

```
with CustomObjectScope({'MyObject':MyObject}):
    layer = Dense(..., kernel_regularizer='MyObject')
    # 保存, 加载等操作将按这个名称来识别自定义对象
```

## HDF5Matrix

[\[source\]](#)

```
keras.utils.HDF5Matrix(datapath, dataset, start=0, end=None, normalizer=None)
```

使用 HDF5 数据集表示，而不是 Numpy 数组。

## 示例

```
x_data = HDF5Matrix('input/file.hdf5', 'data')
model.predict(x_data)
```

提供 `start` 和 `end` 将允许使用数据集的一个切片。

你还可以给出标准化函数（或 `lambda`）（可选）。这将在检索到的每一个数据切片上调用它。

## 参数

- **datapath**: 字符串，HDF5 文件路径。
- **dataset**: 字符串，`datapath`指定的文件中的 HDF5 数据集名称。
- **start**: 整数，所需的指定数据集的切片的开始位置。
- **end**: 整数，所需的指定数据集的切片的结束位置。
- **normalizer**: 在检索数据时调用的函数。

## 返回

一个类似于数组的 HDF5 数据集。

[\[source\]](#)

## Sequence

```
keras.utils.Sequence()
```

用于拟合数据序列的基对象，例如一个数据集。

每一个 `Sequence` 必须实现 `__getitem__` 和 `__len__` 方法。如果你想在迭代之间修改你的数据集，你可以实现 `on_epoch_end`。`__getitem__` 方法应该范围一个完整的批次。

### 注意

`Sequence` 是进行多进程处理的更安全的方法。这种结构保证网络在每个时期每个样本只训练一次，这与生成器不同。

### 示例

```
from skimage.io import imread
from skimage.transform import resize
import numpy as np

# 这里，`x_set` 是图像的路径列表
# 以及 `y_set` 是对应的类别

class CIFAR10Sequence(Sequence):

    def __init__(self, x_set, y_set, batch_size):
        self.x, self.y = x_set, y_set
        self.batch_size = batch_size

    def __len__(self):
        return int(np.ceil(len(self.x) / float(self.batch_size)))

    def __getitem__(self, idx):
        batch_x = self.x[idx * self.batch_size:(idx + 1) * self.batch_size]
        batch_y = self.y[idx * self.batch_size:(idx + 1) * self.batch_size]

        return np.array([
            resize(imread(file_name), (200, 200))
            for file_name in batch_x]), np.array(batch_y)
```

## to\_categorical

```
keras.utils.to_categorical(y, num_classes=None, dtype='float32')
```

将类向量（整数）转换为二进制类矩阵。

例如，用于 `categorical_crossentropy`。

## 参数

- **y**: 需要转换成矩阵的类矢量(从0到num\_classes的整数)。
- **num\_classes**: 总类别数。
- **dtype**: 字符串, 输入所期望的数据类型(`float32`, `float64`, `int32`...)

## 示例

```
# 考虑一组 3 个类 {0, 1, 2} 中的 5 个标签数组:  
> labels  
array([0, 2, 1, 2, 0])  
# `to_categorical` 将其转换为具有尽可能多表示类别数的列的矩阵。  
# 行数保持不变。  
> to_categorical(labels)  
array([[ 1.,  0.,  0.],  
       [ 0.,  0.,  1.],  
       [ 0.,  1.,  0.],  
       [ 0.,  0.,  1.],  
       [ 1.,  0.,  0.]], dtype=float32)
```

## 返回

输入的二进制矩阵表示。

## normalize

```
keras.utils.normalize(x, axis=-1, order=2)
```

标准化一个 Numpy 数组。

## 参数

- **x**: 需要标准化的 Numpy 数组。
- **axis**: 需要标准化的轴。
- **order**: 标准化顺序(例如, 2 表示 L2 规范化)。

## Returns

数组的标准化副本。

## get\_file

```
keras.utils.get_file(fname, origin, untar=False, md5_hash=None,
file_hash=None, cache_subdir='datasets', hash_algorithm='auto', extract=False,
archive_format='auto', cache_dir=None)
```

从一个 URL 下载文件，如果它不存在缓存中。

默认情况下，URL `origin` 处的文件被下载到缓存目录 `~/.keras` 中，放在缓存子目录 `datasets` 中，并命名为 `fname`。文件 `example.txt` 的最终位置为 `~/.keras/datasets/example.txt`。

`tar`, `tar.gz`, `tar.bz`, 以及 `zip` 格式的文件也可以被解压。传递一个哈希值将在下载后校验文件。命令行程序 `shasum` 和 `sha256sum` 可以计算哈希。

## 参数

- **`fname`**: 文件名。如果指定了绝对路径 `/path/to/file.txt`，那么文件将会保存到那个路径。
- **`origin`**: 文件的原始 URL。
- **`untar`**: 由于使用 ‘`extract`’ 而已被弃用。布尔值，是否需要解压文件。
- **`md5_hash`**: 由于使用 ‘`file_hash`’ 而已被弃用。用于校验的 md5 哈希值。
- **`file_hash`**: 下载后的文件的期望哈希字符串。支持 sha256 和 md5 两个哈希算法。
- **`cache_subdir`**: 在 Keras 缓存目录下的保存文件的子目录。如果指定了绝对路径 `/path/to/folder`，则文件将被保存在该位置。
- **`hash_algorithm`**: 选择哈希算法来校验文件。可选的有 ‘`md5`’, ‘`sha256`’, 以及 ‘`auto`’。默认的 ‘`auto`’ 将自动检测所使用的哈希算法。
- **`extract`**: `True` 的话会尝试将解压缩存档文件，如 `tar` 或 `zip`。
- **`archive_format`**: 尝试提取文件的存档格式。可选的有 ‘`auto`’, ‘`tar`’, ‘`zip`’, 以及 `None`。‘`tar`’ 包含 `tar`, `tar.gz`, 和 `tar.bz` 文件。默认 ‘`auto`’ 为 [‘`tar`’, ‘`zip`’]。 `None` 或 空列表将返回未找到任何匹配。ke xu az z’`auto`’, ‘`tar`’, ‘`zip`’, and `None`.
- **`cache_dir`**: 存储缓存文件的位置，为 `None` 时默认为 [Keras 目录](#).

## 返回

下载的文件的路径。

## print\_summary

```
keras.utils.print_summary(model, line_length=None, positions=None,
print_fn=None)
```

打印模型概况。

## 参数

- **model**: Keras 模型实例。
- **line\_length**: 打印的每行的总长度 (例如, 设置此项以使其显示适应不同的终端窗口大小)。
- **positions**: 每行中日志元素的相对或绝对位置。如果未提供, 默认为  
[ .33, .55, .67, 1.]。
- **print\_fn**: 需要使用的打印函数。它将在每一行概述时调用。您可以将其设置为自定义函数以捕获字符串概述。默认为 `print` (打印到标准输出)。

## plot\_model

```
keras.utils.plot_model(model, to_file='model.png', show_shapes=False,  
show_layer_names=True, rankdir='TB', expand_nested=False, dpi=96)
```

将 Keras 模型转换为 dot 格式并保存到文件中。

## 参数

- **model**: 一个 Keras 模型实例。
- **to\_file**: 绘制图像的文件名。
- **show\_shapes**: 是否显示尺寸信息。
- **show\_layer\_names**: 是否显示层的名称。
- **rankdir**: 传递给 PyDot 的 `rankdir` 参数, 一个指定绘图格式的字符串: 'TB' 创建一个垂直绘图; 'LR' 创建一个水平绘图。
- **expand\_nested**: 是否扩展嵌套模型为聚类。
- **dpi**: 点 DPI。

## 返回

如果安装了 Jupyter, 则返回一个 Jupyter notebook Image 对象。这样可以在 notebook 中在线显示模型图。

## multi\_gpu\_model

```
keras.utils.multi_gpu_model(model, gpus=None, cpu_merge=True,  
cpu_relocation=False)
```

将模型复制到不同的 GPU 上。

具体来说，该功能实现了单机多 GPU 数据并行性。它的工作原理如下：

- 将模型的输入分成多个子批次。
- 在每个子批次上应用模型副本。每个模型副本都在专用 GPU 上执行。
- 将结果（在 CPU 上）连接成一个大批量。

例如，如果你的 `batch_size` 是 64，且你使用 `gpus=2`，那么我们将把输入分为两个 32 个样本的子批次，在 1 个 GPU 上处理 1 个子批次，然后返回完整批次的 64 个处理过的样本。

这实现了多达 8 个 GPU 的准线性加速。

此功能目前仅适用于 TensorFlow 后端。

## 参数

- **model**: 一个 Keras 模型实例。为了避免 OOM 错误，该模型可以建立在 CPU 上，详见下面的使用样例。
- **gpus**: 整数  $\geq 2$  或整数列表，创建模型副本的 GPU 数量，或 GPU ID 的列表。
- **cpu\_merge**: 一个布尔值，用于标识是否强制合并 CPU 范围内的模型权重。
- **cpu\_relocation**: 一个布尔值，用来确定是否在 CPU 的范围内创建模型的权重。如果模型没有在任何一个设备范围内定义，您仍然可以通过激活这个选项来拯救它。

## 返回

一个 Keras Model 实例，它可以像初始 `model` 参数一样使用，但它将工作负载分布在多个 GPU 上。

## 示例

### 例 1 - 训练在 CPU 上合并权重的模型

```
import tensorflow as tf
from keras.applications import Xception
from keras.utils import multi_gpu_model
import numpy as np

num_samples = 1000
height = 224
width = 224
num_classes = 1000

# 实例化基础模型（或者「模版」模型）。
# 我们推荐在 CPU 设备范围内做此操作，
# 这样模型的权重就会存储在 CPU 内存中。
# 否则它们会存储在 GPU 上，而完全被共享。
with tf.device('/cpu:0'):
```

```

model = Xception(weights=None,
                   input_shape=(height, width, 3),
                   classes=num_classes)

# 复制模型到 8 个 GPU 上。
# 这假设你的机器有 8 个可用 GPU。
parallel_model = multi_gpu_model(model, gpus=8)
parallel_model.compile(loss='categorical_crossentropy',
                       optimizer='rmsprop')

# 生成虚拟数据
x = np.random.random((num_samples, height, width, 3))
y = np.random.random((num_samples, num_classes))

# 这个 `fit` 调用将分布在 8 个 GPU 上。
# 由于 batch size 是 256，每个 GPU 将处理 32 个样本。
parallel_model.fit(x, y, epochs=20, batch_size=256)

# 通过模版模型存储模型（共享相同权重）：
model.save('my_model.h5')

```

#### 例 2 - 训练在 CPU 上利用 cpu\_relocation 合并权重的模型

```

...
# 不需要更改模型定义的设备范围：
model = Xception(weights=None, ...)

try:
    parallel_model = multi_gpu_model(model, cpu_relocation=True)
    print("Training using multiple GPUs...")
except ValueError:
    parallel_model = model
    print("Training using single GPU or CPU...")
parallel_model.compile(...)

...

```

#### 例 3 - 训练在 GPU 上合并权重的模型（建议用于 NV-link）

```

...
# 不需要更改模型定义的设备范围：
model = Xception(weights=None, ...)

try:
    parallel_model = multi_gpu_model(model, cpu_merge=False)
    print("Training using multiple GPUs...")
except:
    parallel_model = model
    print("Training using single GPU or CPU...")

parallel_model.compile(...)

...

```

#### 关于模型保存

要保存多 GPU 模型，请通过模板模型（传递给 `multi_gpu_model` 的参数）调用 `.save(fname)` 或 `.save_weights(fname)` 以进行存储，而不是通过 `multi_gpu_model` 返回的模型。



# 关于 Github Issues 和 Pull Requests

找到一个漏洞？有一个新的功能建议？想要对代码库做出贡献？请务必先阅读这些。

## 漏洞报告

你的代码不起作用，你确定问题在于Keras？请按照以下步骤报告错误。

1. 你的漏洞可能已经被修复了。确保更新到目前的Keras master分支，以及最新的Theano/TensorFlow/CNTK master分支。轻松更新Theano的方法：`pip install git+git://github.com/Theano/Theano.git --upgrade`
2. 搜索相似问题。确保在搜索已经解决的Issue时删除`is:open`标签。有可能已经有人遇到了这个漏洞。同时记得检查Keras [FAQ](#)。仍然有问题？在Github上开一个Issue，让我们知道。
3. 确保你向我们提供了有关你的配置的有用信息：什么操作系统？什么Keras后端？你是否在GPU上运行，Cuda和cuDNN的版本是多少？GPU型号是什么？
4. 为我们提供一个脚本来重现这个问题。该脚本应该可以按原样运行，并且不应该要求下载外部数据（如果需要在某些测试数据上运行模型，请使用随机生成的数据）。我们建议你使用Github Gists来张贴你的代码。任何无法重现的问题都会被关闭。
5. 如果可能的话，自己动手修复这个漏洞 - 如果可以的话！

你提供的信息越多，我们就越容易验证存在错误，并且我们可以采取更快的行动。如果你想快速解决你的问题，尊许上述步骤操作是至关重要的。

## 请求新功能

你也可以使用[TensorFlow Github issues](#)来请求你希望在Keras中看到的功能，或者在Keras API中的更改。

1. 提供你想要的功能的清晰和详细的解释，以及为什么添加它很重要。请记住，我们需要的功能是对于大多数用户而言的，不仅仅是一小部分人。如果你只是针对少数用户，请考虑为Keras编写附加库。对Keras来说，避免臃肿的API和代码库是至关重要的。
2. 提供代码片段，演示您所需的API并说明您的功能的用例。当然，在这一点上你不需要写任何真正的代码！

3. 讨论完需要在 `tf.keras` 中添加该功能后，你可以选择尝试提一个 Pull Request。如果你完全可以，开始写一些代码。相比时间上，我们总是有更多的工作要做。如果你可以写一些代码，那么这将加速这个过程。

## 请求贡献代码

在这个板块我们会列出当前需要添加的出色的问题和新功能。如果你想要为 Keras 做贡献，这就是可以开始的地方。

## Pull Requests 合并请求

### 我应该在哪里提交我的合并请求？

注意：

我们不再会向多后端 Keras 添加新功能（我们仅仅修复漏洞），因为我们正重新将开发精力投入到 `tf.keras` 上。如果你仍然像提交新的功能申请，请直接将它提交到 TensorFlow 仓库的 `tf.keras` 中。

1. **Keras 改进与漏洞修复**，请到 [Keras master 分支](#)。
2. **测试新功能**，例如网络层和数据集，请到 [keras-contrib](#)。除非它是一个在 [Requests for Contributions](#) 中列出的新功能，它属于 Keras 的核心部分。如果你觉得你的功能属于 Keras 核心，你可以提交一个设计文档，来解释你的功能，并争取它（请看以下解释）。

请注意任何有关 **代码风格**（而不是修复修复，改进文档或添加新功能）的 PR 都会被拒绝。

以下是提交你的改进的快速指南：

1. 如果你的 PR 介绍了功能的改变，确保你从撰写设计文档并将其发给 Keras 邮件列表开始，以讨论是否应该修改，以及如何处理。这将拯救你于 PR 关闭。当然，如果你的 PR 只是一个简单的漏洞修复，那就不需要这样做。撰写与提交设计文档的过程如下所示：
  - 从这个 [Google 文档模版](#) 开始，将它复制为一个新的 Google 文档。
  - 填写内容。注意你需要插入代码样例。要插入代码，请使用 Google 文档插件，例如 [CodePretty] (<https://chrome.google.com/webstore/detail/code-pretty/igjbncgfgnfpbnifnnlcmjfbnidkndnh?hl=en>) (有许多可用的插件)。
  - 将共享设置为「每个有链接的人都可以发表评论」。
  - 将文档发给 `keras-users@googlegroups.com`，主题从 `[API DESIGN REVIEW]` (全大写) 开始，这样我们才会注意到它。

- 等待评论，回复评论。必要时修改提案。
  - 该提案最终将被批准或拒绝。一旦获得批准，您可以发出合并请求或要求他人撰写合并请求。
2. 撰写代码（或者让别人写）。这是最难的一部分。
3. 确保你引入的任何新功能或类都有适当的文档。确保你触摸的任何代码仍具有最新的文档。  
**应该严格遵循 Docstring 风格**。尤其是，它们应该在 MarkDown 中格式化，并且应该有 `Arguments`, `Returns`, `Raises` 部分（如果适用）。查看代码示例中的其他文档以做参考。
4. 撰写测试。你的代码应该有完整的单元测试覆盖。如果你想看到你的 PR 迅速合并，这是至关重要的。
5. 在本地运行测试套件。这很简单：在 Keras 目录下，直接运行：`py.test tests/`。
  - 您还需要安装测试包：`pip install -e .[tests]`。
6. 确保通过所有测试：
- 使用 Theano 后端，Python 2.7 和 Python 3.5。确保你有 Theano 的开发版本。
  - 使用 TensorFlow 后端，Python 2.7 和 Python 3.5。确保你有 TensorFlow 的开发版本。
  - 使用 CNTK 后端，Python 2.7 和 Python 3.5。确保你有 CNTK 的开发版本。
7. 我们使用 PEP8 语法约定，但是当涉及到行长时，我们不是教条式的。尽管如此，确保你的行保持合理的大小。为了让您的生活更轻松，我们推荐使用 PEP8 linter：
  - 安装 PEP8 包：`pip install pep8 pytest-pep8 autopep8`
  - 运行独立的 PEP8 检查：`py.test --pep8 -m pep8`
  - 你可以通过运行这个命令自动修复一些 PEP8 错误：`autopep8 -i --select <errors> <FILENAME>`。例如：  
`autopep8 -i --select E128 tests/keras/backend/test_backends.py`
8. 提交时，请使用适当的描述性提交消息。
9. 更新文档。如果引入新功能，请确保包含演示新功能用法的代码片段。
10. 提交你的 PR。如果你的更改已在之前的讨论中获得批准，并且你有完整（并通过）的单元测试以及正确的 docstring/文档，则你的 PR 可能会立即合并。

## 添加新的样例

即使你不贡献 Keras 源代码，如果你有一个简洁而强大的 Keras 应用，请考虑将它添加到我们的样例集合中。[现有的例子](#)展示惯用的 Keras 代码：确保保持自己的脚本具有相同的风格。



# 实现一个用来执行加法的序列到序列学习模型

输入: “535+61”

输出: “596”

使用重复的标记字符（空格）处理填充。

输入可以选择性地反转，它被认为可以提高许多任务的性能，例如：[Learning to Execute](#) 以及 [Sequence to Sequence Learning with Neural Networks](#)。

从理论上讲，它引入了源和目标之间的短期依赖关系。

两个反转的数字 + 一个 LSTM 层（128个隐藏单元），在 55 个 epochs 后，5k 的训练样本取得了 99% 的训练/测试准确率。

三个反转的数字 + 一个 LSTM 层（128个隐藏单元），在 100 个 epochs 后，50k 的训练样本取得了 99% 的训练/测试准确率。

四个反转的数字 + 一个 LSTM 层（128个隐藏单元），在 20 个 epochs 后，400k 的训练样本取得了 99% 的训练/测试准确率。

五个反转的数字 + 一个 LSTM 层（128个隐藏单元），在 30 个 epochs 后，550k 的训练样本取得了 99% 的训练/测试准确率。

```
from __future__ import print_function
from keras.models import Sequential
from keras import layers
import numpy as np
from six.moves import range

class CharacterTable(object):
    """给定一组字符：
    + 将它们编码为 one-hot 整数表示
    + 将 one-hot 或整数表示解码为字符输出
    + 将一个概率向量解码为字符输出
    """
    def __init__(self, chars):
        """初始化字符表。

        # 参数:
        chars: 可以出现在输入中的字符。
        """
        self.chars = sorted(set(chars))
        self.char_indices = dict((c, i) for i, c in enumerate(self.chars))
        self.indices_char = dict((i, c) for i, c in enumerate(self.chars))

    def encode(self, C, num_rows):
```

```

    """给定字符串 C 的 one-hot 编码。
    # 参数
        C: 需要编码的字符串。
        num_rows: 返回的 one-hot 编码的行数。
            这用来保证每个数据的行数相同。
    """
    x = np.zeros((num_rows, len(self.chars)))
    for i, c in enumerate(C):
        x[i, self.char_indices[c]] = 1
    return x

def decode(self, x, calc_argmax=True):
    """将给定的向量或 2D array 解码为它们的字符输出。

    # 参数
        x: 一个向量或 2D 概率数组或 one-hot 表示,
            或 字符索引的向量 (如果 `calc_argmax=False` )。
        calc_argmax: 是否根据最大概率来找到字符, 默认为 `True`。
    """
    if calc_argmax:
        x = x.argmax(axis=-1)
    return ''.join(self.indices_char[x] for x in x)

class colors:
    ok = '\033[92m'
    fail = '\033[91m'
    close = '\033[0m'

# 模型和数据的参数
TRAINING_SIZE = 50000
DIGITS = 3
REVERSE = True

# 输入的最大长度是 'int+int' (例如, '345+678')。int 的最大长度为 DIGITS。
MAXLEN = DIGITS + 1 + DIGITS

# 所有的数字, 加上符号, 以及用于填充的空格。
chars = '0123456789+ '
ctable = CharacterTable(chars)

questions = []
expected = []
seen = set()
print('Generating data...')
while len(questions) < TRAINING_SIZE:
    f = lambda: int(''.join(np.random.choice(list('0123456789')) for i in range(np.random.randint(1, DIGITS + 1))))
    a, b = f(), f()
    # 跳过任何已有的加法问题
    # 同事跳过任何 x+Y == Y+x 的情况(即排序)。
    key = tuple(sorted((a, b)))
    if key in seen:
        continue
    seen.add(key)
    # 利用空格填充, 是的长度始终为 MAXLEN。

```

```

q = '{}+{}'.format(a, b)
query = q + ' ' * (MAXLEN - len(q))
ans = str(a + b)
# 答案可能的最长长度为 DIGITS + 1。
ans += ' ' * (DIGITS + 1 - len(ans))
if REVERSE:
    # 反转查询，例如，'12+345' 变成 '543+21'.
    # (注意用于填充的空格)
    query = query[::-1]
questions.append(query)
expected.append(ans)
print('Total addition questions:', len(questions))

print('Vectorization...')
x = np.zeros((len(questions), MAXLEN, len(chars)), dtype=np.bool)
y = np.zeros((len(questions), DIGITS + 1, len(chars)), dtype=np.bool)
for i, sentence in enumerate(questions):
    x[i] = ctable.encode(sentence, MAXLEN)
for i, sentence in enumerate(expected):
    y[i] = ctable.encode(sentence, DIGITS + 1)

# 混洗 (x, y)，因为 x 的后半段几乎都是比较大的数字。
indices = np.arange(len(y))
np.random.shuffle(indices)
x = x[indices]
y = y[indices]

# 显式地分离出 10% 的训练数据作为验证集。
split_at = len(x) - len(x) // 10
(x_train, x_val) = x[:split_at], x[split_at:]
(y_train, y_val) = y[:split_at], y[split_at:]

print('Training Data:')
print(x_train.shape)
print(y_train.shape)

print('Validation Data:')
print(x_val.shape)
print(y_val.shape)

# 可以尝试更改为 GRU，或 SimpleRNN。
RNN = layers.LSTM
HIDDEN_SIZE = 128
BATCH_SIZE = 128
LAYERS = 1

print('Build model...')
model = Sequential()
# 利用 RNN 将输入序列「编码」为一个 HIDDEN_SIZE 长度的输出向量。
# 注意：在输入序列具有可变长度的情况下，
# 使用 input_shape=(None, num_feature).
model.add(RNN(HIDDEN_SIZE, input_shape=(MAXLEN, len(chars))))
# 作为解码器 RNN 的输入，为每个时间步重复地提供 RNN 的最后输出。
# 重复 'DIGITS + 1' 次，因为它是最大输出长度。
# 例如，当 DIGITS=3，最大输出为 999+999=1998。
model.add(layers.RepeatVector(DIGITS + 1))
# 解码器 RNN 可以是多个堆叠的层，或一个单独的层。

```

```

for _ in range(LAYERS):
    # 通过设置 return_sequences 为 True, 将不仅返回最后一个输出, 而是返回目前的所有输出, 形式为(num_samples, timesteps, output_dim)。
    # 这是必须的, 因为后面的 TimeDistributed 需要第一个维度是时间步。
    model.add(RNN(HIDDEN_SIZE, return_sequences=True))

# 将全连接层应用于输入的每个时间片。
# 对于输出序列的每一步, 决定应选哪个字符。
model.add(layers.TimeDistributed(layers.Dense(len(chars),
activation='softmax')))
model.compile(loss='categorical_crossentropy',
optimizer='adam',
metrics=[ 'accuracy' ])
model.summary()

# 训练模型, 并在每一代显示验证数据的预测。
for iteration in range(1, 200):
    print()
    print('-' * 50)
    print('Iteration', iteration)
    model.fit(x_train, y_train,
               batch_size=BATCH_SIZE,
               epochs=1,
               validation_data=(x_val, y_val))
    # 从随机验证集中选择 10 个样本, 以便我们可以看到错误。
    for i in range(10):
        ind = np.random.randint(0, len(x_val))
        rowx, rowy = x_val[np.array([ind])], y_val[np.array([ind])]
        preds = model.predict_classes(rowx, verbose=0)
        q = ctable.decode(rowx[0])
        correct = ctable.decode(rowy[0])
        guess = ctable.decode(preds[0], calc_argmax=False)
        print('Q', q[::-1] if REVERSE else q, end=' ')
        print('T', correct, end=' ')
        if correct == guess:
            print(colors.ok + '☑' + colors.close, end=' ')
        else:
            print(colors.fail + '☒' + colors.close, end=' ')
        print(guess)

```



# 本示例演示了如何为 Keras 编写自定义网络层。

我们构建了一个称为 ‘Antirectifier’ 的自定义激活层，该层可以修改通过它的张量的形状。我们需要指定两个方法: `compute_output_shape` 和 `call`。

注意，相同的结果也可以通过 Lambda 层取得。

我们的自定义层是使用 Keras 后端 (`K`) 中的基元编写的，因而代码可以在 TensorFlow 和 Theano 上运行。

```
from __future__ import print_function
import keras
from keras.models import Sequential
from keras import layers
from keras.datasets import mnist
from keras import backend as K

class Antirectifier(layers.Layer):
    '''这是样本级的 L2 标准化与输入的正负部分串联的组合。
    结果是两倍于输入样本的样本张量。

    它可以用于替代 ReLU。

    # 输入尺寸
    2D 张量，尺寸为 (samples, n)

    # 输出尺寸
    2D 张量，尺寸为 (samples, 2*n)

    # 理论依据
    在应用 ReLU 时，假设先前输出的分布接近于 0 的中心，
    那么将丢弃一半的输入。这是非常低效的。

    Antirectifier 允许像 ReLU 一样返回全正输出，而不会丢弃任何数据。

    在 MNIST 上进行的测试表明，Antirectifier 可以训练参数少两倍但具
    有与基于 ReLU 的等效网络相当的分类精度的网络。
    '''

    def compute_output_shape(self, input_shape):
        shape = list(input_shape)
        assert len(shape) == 2  # 仅对 2D 张量有效
        shape[-1] *= 2
        return tuple(shape)

    def call(self, inputs):
        inputs -= K.mean(inputs, axis=1, keepdims=True)
        inputs = K.l2_normalize(inputs, axis=1)
        pos = K.relu(inputs)
        neg = K.relu(-inputs)
```

```
    return K.concatenate([pos, neg], axis=1)

# 全局参数
batch_size = 128
num_classes = 10
epochs = 40

# 切分为训练和测试的数据
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# 将类向量转化为二进制类矩阵
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# 构建模型
model = Sequential()
model.add(layers.Dense(256, input_shape=(784,)))
model.add(Antirectifier())
model.add(layers.Dropout(0.1))
model.add(layers.Dense(256))
model.add(Antirectifier())
model.add(layers.Dropout(0.1))
model.add(layers.Dense(num_classes))
model.add(layers.Activation('softmax'))

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

# 训练模型
model.fit(x_train, y_train,
           batch_size=batch_size,
           epochs=epochs,
           verbose=1,
           validation_data=(x_test, y_test))

# 接下来，与具有 2 倍大的密集层
# 和 ReLU 的等效网络进行比较
```



# 基于故事和问题训练两个循环神经网络。

两者的合并向量将用于回答一系列 bAbI 任务。

这些结果与 Weston 等人提供的 LSTM 模型的结果相当：[Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks](#)。

Task Number	FB LSTM Baseline	Keras QA
QA1 - Single Supporting Fact	50	52.1
QA2 - Two Supporting Facts	20	37.0
QA3 - Three Supporting Facts	20	20.5
QA4 - Two Arg. Relations	61	62.9
QA5 - Three Arg. Relations	70	61.9
QA6 - yes/No Questions	48	50.7
QA7 - Counting	49	78.9
QA8 - Lists/Sets	45	77.2
QA9 - Simple Negation	64	64.0
QA10 - Indefinite Knowledge	44	47.7
QA11 - Basic Coreference	72	74.9
QA12 - Conjunction	74	76.4
QA13 - Compound Coreference	94	94.4
QA14 - Time Reasoning	27	34.8
QA15 - Basic Deduction	21	32.4

Task Number	FB LSTM Baseline	Keras QA
QA16 - Basic Induction	23	50.6
QA17 - Positional Reasoning	51	49.1
QA18 - Size Reasoning	52	90.8
QA19 - Path Finding	8	9.0
QA20 - Agent's Motivations	91	90.7

有关 bAbI 项目的相关资源，请参考: <https://research.facebook.com/researchers/1543934539189348>

## 注意

- 使用默认的单词、句子和查询向量尺寸，GRU 模型得到了以下效果：
  - 20 轮迭代后，在 QA1 上达到了 52.1% 的测试准确率（在 CPU 上每轮迭代 2 秒）；
  - 20 轮迭代后，在 QA2 上达到了 37.0% 的测试准确率（在 CPU 上每轮迭代 16 秒）。
- 相比之下，Facebook 的论文中 LSTM baseline 的准确率分别是 50% 和 20%。
- 这个任务并不是笼统地单独去解析问题。这应该可以提高准确率，且是合并两个 RNN 的一次较好实践。
- 故事和问题的 RNN 之间不共享词向量（词嵌入）。
- 注意观察 1000 个训练样本 (en-10k) 到 10,000 个的准确度如何变化。使用 1000 是为了与原始论文进行对比。
- 尝试使用 GRU, LSTM 和 JZS1-3，因为它们会产生微妙的不同结果。
- 长度和噪声（即「无用」的故事内容）会影响 LSTM/GRU 提供正确答案的能力。在只提供事实的情况下，这些 RNN 可以在许多任务上达到 100% 的准确性。使用注意力过程的记忆网络和神经网络可以有效地搜索这些噪声以找到相关的语句，从而大大提高性能。这在 QA2 和 QA3 上变得尤为明显，两者都远远显著于 QA1。

```

from __future__ import print_function
from functools import reduce
import re
import tarfile

import numpy as np

from keras.utils.data_utils import get_file

```

```

from keras.layers.embeddings import Embedding
from keras import layers
from keras.layers import recurrent
from keras.models import Model
from keras.preprocessing.sequence import pad_sequences

def tokenize(sent):
    '''返回包含标点符号的句子的标记。

    >>> tokenize('Bob dropped the apple. Where is the apple?')
    ['Bob', 'dropped', 'the', 'apple', '.', 'Where', 'is', 'the', 'apple',
    '?']
    ...
    return [x.strip() for x in re.split(r'(\W+)', sent) if x.strip()]

def parse_stories(lines, only_supporting=False):
    '''解析 bAbi 任务格式中提供的故事

    如果 only_supporting 为 true,
    则只保留支持答案的句子。
    ...
    data = []
    story = []
    for line in lines:
        line = line.decode('utf-8').strip()
        nid, line = line.split(' ', 1)
        nid = int(nid)
        if nid == 1:
            story = []
        if '\t' in line:
            q, a, supporting = line.split('\t')
            q = tokenize(q)
            if only_supporting:
                # 只选择相关的子故事
                supporting = map(int, supporting.split())
                substory = [story[i - 1] for i in supporting]
            else:
                # 提供所有子故事
                substory = [x for x in story if x]
            data.append((substory, q, a))
            story.append('')
        else:
            sent = tokenize(line)
            story.append(sent)
    return data

def get_stories(f, only_supporting=False, max_length=None):
    '''给定文件名, 读取文件, 检索故事,
    然后将句子转换为一个独立故事。

    如果提供了 max_length,
    任何长于 max_length 的故事都将被丢弃。
    ...
    data = parse_stories(f.readlines(), only_supporting=only_supporting)

```

```

flatten = lambda data: reduce(lambda x, y: x + y, data)
data = [(flatten(story), q, answer) for story, q, answer in data
         if not max_length or len(flatten(story)) < max_length]
return data

def vectorize_stories(data, word_idx, story maxlen, query maxlen):
    xs = []
    xqs = []
    ys = []
    for story, query, answer in data:
        x = [word_idx[w] for w in story]
        xq = [word_idx[w] for w in query]
        # 不要忘记索引 0 已被保留
        y = np.zeros(len(word_idx) + 1)
        y[word_idx[answer]] = 1
        xs.append(x)
        xqs.append(xq)
        ys.append(y)
    return (pad_sequences(xs, maxlen=story maxlen),
            pad_sequences(xqs, maxlen=query maxlen), np.array(ys))

RNN = recurrent.LSTM
EMBED_HIDDEN_SIZE = 50
SENT_HIDDEN_SIZE = 100
QUERY_HIDDEN_SIZE = 100
BATCH_SIZE = 32
EPOCHS = 20
print('RNN / Embed / Sent / Query = {}, {}, {}, {}'.format(RNN,
                                                               EMBED_HIDDEN_SIZE,
                                                               SENT_HIDDEN_SIZE,
                                                               QUERY_HIDDEN_SIZE))

try:
    path = get_file('babi-tasks-v1-2.tar.gz',
                    origin='https://s3.amazonaws.com/text-datasets/'
                           'babi_tasks_1-20_v1-2.tar.gz')
except:
    print('Error downloading dataset, please download it manually:\n'
          '$ wget http://www.thespermwhale.com/jaseweston/babi/'
          'tasks_1-20_v1-2'
          '.tar.gz\n'
          '$ mv tasks_1-20_v1-2.tar.gz ~/.keras/datasets/babi-tasks-'
          'v1-2.tar.gz')
    raise

# 默认 QA1 任务, 1000 样本
# challenge = 'tasks_1-20_v1-2/en/qa1_single-supporting-fact_{}.txt'
# QA1 任务, 10,000 样本
# challenge = 'tasks_1-20_v1-2/en-10k/qa1_single-supporting-fact_{}.txt'
# QA2 任务, 1000 样本
challenge = 'tasks_1-20_v1-2/en/qa2_two-supporting-facts_{}.txt'
# QA2 任务, 10,000 样本
# challenge = 'tasks_1-20_v1-2/en-10k/qa2_two-supporting-facts_{}.txt'
with tarfile.open(path) as tar:
    train = get_stories(tar.extractfile(challenge.format('train')))
    test = get_stories(tar.extractfile(challenge.format('test')))
```

```

vocab = set()
for story, q, answer in train + test:
    vocab |= set(story + q + [answer])
vocab = sorted(vocab)

# 保留 0 以留作 pad_sequences 进行 masking
vocab_size = len(vocab) + 1
word_idx = dict((c, i + 1) for i, c in enumerate(vocab))
story_maxlen = max(map(len, (x for x, _, _ in train + test)))
query_maxlen = max(map(len, (x for _, x, _ in train + test)))

x, xq, y = vectorize_stories(train, word_idx, story_maxlen, query_maxlen)
tx, txq, ty = vectorize_stories(test, word_idx, story_maxlen, query_maxlen)

print('vocab = {}'.format(vocab))
print('x.shape = {}'.format(x.shape))
print('xq.shape = {}'.format(xq.shape))
print('y.shape = {}'.format(y.shape))
print('story_maxlen, query_maxlen = {}, {}'.format(story_maxlen,
query_maxlen))

print('Build model...')

sentence = layers.Input(shape=(story_maxlen,), dtype='int32')
encoded_sentence = layers.Embedding(vocab_size, EMBED_HIDDEN_SIZE)(sentence)
encoded_sentence = RNN(SENT_HIDDEN_SIZE)(encoded_sentence)

question = layers.Input(shape=(query_maxlen,), dtype='int32')
encoded_question = layers.Embedding(vocab_size, EMBED_HIDDEN_SIZE)(question)
encoded_question = RNN(QUERY_HIDDEN_SIZE)(encoded_question)

merged = layers.concatenate([encoded_sentence, encoded_question])
preds = layers.Dense(vocab_size, activation='softmax')(merged)

model = Model([sentence, question], preds)
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

print('Training')
model.fit([x, xq], y,
          batch_size=BATCH_SIZE,
          epochs=EPOCHS,
          validation_split=0.05)

print('Evaluation')
loss, acc = model.evaluate([tx, txq], ty,
                           batch_size=BATCH_SIZE)
print('Test loss / test accuracy = {:.4f} / {:.4f}'.format(loss, acc))

```



# 在 bAbI 数据集上训练一个记忆网络。

参考文献：

- Jason Weston, Antoine Bordes, Sumit Chopra, Tomas Mikolov, Alexander M. Rush, “Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks”
- Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, Rob Fergus, “End-To-End Memory Networks”

120 轮迭代后，在 ‘single\_supporting\_fact\_10k’ 任务上达到了 98.6% 的准确率。每轮迭代时间：3s on CPU (core i7)。

```
from __future__ import print_function

from keras.models import Sequential, Model
from keras.layers.embeddings import Embedding
from keras.layers import Input, Activation, Dense, Permute, Dropout
from keras.layers import add, dot, concatenate
from keras.layers import LSTM
from keras.utils.data_utils import get_file
from keras.preprocessing.sequence import pad_sequences
from functools import reduce
import tarfile
import numpy as np
import re

def tokenize(sent):
    '''返回包含标点符号的句子的标记。

    >>> tokenize('Bob dropped the apple. Where is the apple?')
    ['Bob', 'dropped', 'the', 'apple', '.', 'Where', 'is', 'the', 'apple',
    '?']
    ...
    return [x.strip() for x in re.split(r'(\W+)?', sent) if x.strip()]

def parse_stories(lines, only_supporting=False):
    '''解析 bAbi 任务格式中提供的故事

    如果 only_supporting 为 true,
    则只保留支持答案的句子。
    ...
    data = []
    story = []
    for line in lines:
        line = line.decode('utf-8').strip()
        nid, line = line.split(' ', 1)
        nid = int(nid)
        if nid == 1:
```

```

        story = []
    if '\t' in line:
        q, a, supporting = line.split('\t')
        q = tokenize(q)
        if only_supporting:
            # 只选择相关的子故事
            supporting = map(int, supporting.split())
            substory = [story[i - 1] for i in supporting]
        else:
            # 提供所有子故事
            substory = [x for x in story if x]
        data.append((substory, q, a))
        story.append('')
    else:
        sent = tokenize(line)
        story.append(sent)
return data

def get_stories(f, only_supporting=False, max_length=None):
    '''给定文件名, 读取文件, 检索故事,
    然后将句子转换为一个独立故事。

    如果提供了 max_length,
    任何长于 max_length 的故事都将被丢弃。
    '''

    data = parse_stories(f.readlines(), only_supporting=only_supporting)
    flatten = lambda data: reduce(lambda x, y: x + y, data)
    data = [(flatten(story), q, answer) for story, q, answer in data
            if not max_length or len(flatten(story)) < max_length]
    return data

def vectorize_stories(data):
    inputs, queries, answers = [], [], []
    for story, query, answer in data:
        inputs.append([word_idx[w] for w in story])
        queries.append([word_idx[w] for w in query])
        answers.append(word_idx[answer])
    return (pad_sequences(inputs, maxlen=story_maxlen),
            pad_sequences(queries, maxlen=query_maxlen),
            np.array(answers))

try:
    path = get_file('babi-tasks-v1-2.tar.gz',
                    origin='https://s3.amazonaws.com/text-datasets/'
                    'babi_tasks_1-20_v1-2.tar.gz')
except:
    print('Error downloading dataset, please download it manually:\n'
          '$ wget http://www.thespermwhale.com/jaseweston/babi/'
          'tasks_1-20_v1-2'
          '.tar.gz\n'
          '$ mv tasks_1-20_v1-2.tar.gz ~/.keras/datasets/babi-tasks-'
          'v1-2.tar.gz')
    raise

```

```

challenges = {
    # QA1 任务, 10,000 样本
    'single_supporting_fact_10k': 'tasks_1-20_v1-2/en-10k/qa1_'
                                    'single-supporting-fact_{}.txt',
    # QA2 任务, 1000 样本
    'two_supporting_facts_10k': 'tasks_1-20_v1-2/en-10k/qa2_'
                                'two-supporting-facts_{}.txt',
}
challenge_type = 'single_supporting_fact_10k'
challenge = challenges[challenge_type]

print('Extracting stories for the challenge:', challenge_type)
with tarfile.open(path) as tar:
    train_stories = get_stories(tar.extractfile(challenge.format('train')))
    test_stories = get_stories(tar.extractfile(challenge.format('test')))

vocab = set()
for story, q, answer in train_stories + test_stories:
    vocab |= set(story + q + [answer])
vocab = sorted(vocab)

# 保留 0 以留作 pad_sequences 进行 masking
vocab_size = len(vocab) + 1
story_maxlen = max(map(len, (x for x, _, _ in train_stories + test_stories)))
query_maxlen = max(map(len, (x for _, x, _ in train_stories + test_stories)))

print('-')
print('Vocab size:', vocab_size, 'unique words')
print('Story max length:', story_maxlen, 'words')
print('Query max length:', query_maxlen, 'words')
print('Number of training stories:', len(train_stories))
print('Number of test stories:', len(test_stories))
print('-')
print('Here\'s what a "story" tuple looks like (input, query, answer):')
print(train_stories[0])
print('-')
print('Vectorizing the word sequences...')

word_idx = dict((c, i + 1) for i, c in enumerate(vocab))
inputs_train, queries_train, answers_train = vectorize_stories(train_stories)
inputs_test, queries_test, answers_test = vectorize_stories(test_stories)

print('-')
print('inputs: integer tensor of shape (samples, max_length)')
print('inputs_train shape:', inputs_train.shape)
print('inputs_test shape:', inputs_test.shape)
print('-')
print('queries: integer tensor of shape (samples, max_length)')
print('queries_train shape:', queries_train.shape)
print('queries_test shape:', queries_test.shape)
print('-')
print('answers: binary (1 or 0) tensor of shape (samples, vocab_size)')
print('answers_train shape:', answers_train.shape)
print('answers_test shape:', answers_test.shape)
print('-')
print('Compiling...')

```

```

# 占位符
input_sequence = Input((story_maxlen,))
question = Input((query_maxlen,))

# 编码器
# 将输入序列编码为向量的序列
input_encoder_m = Sequential()
input_encoder_m.add(Embedding(input_dim=vocab_size,
                             output_dim=64))
input_encoder_m.add(Dropout(0.3))
# 输出: (samples, story_maxlen, embedding_dim)

# 将输入编码为的向量的序列 (向量尺寸为 query_maxlen)
input_encoder_c = Sequential()
input_encoder_c.add(Embedding(input_dim=vocab_size,
                             output_dim=query_maxlen))
input_encoder_c.add(Dropout(0.3))
# 输出: (samples, story_maxlen, query_maxlen)

# 将问题编码为向量的序列
question_encoder = Sequential()
question_encoder.add(Embedding(input_dim=vocab_size,
                             output_dim=64,
                             input_length=query_maxlen))
question_encoder.add(Dropout(0.3))
# 输出: (samples, query_maxlen, embedding_dim)

# 编码输入序列和问题 (均已索引化) 为密集向量的序列
input_encoded_m = input_encoder_m(input_sequence)
input_encoded_c = input_encoder_c(input_sequence)
question_encoded = question_encoder(question)

# 计算第一个输入向量和问题向量序列的『匹配』 ('match')
# 尺寸: `(samples, story_maxlen, query_maxlen)`
match = dot([input_encoded_m, question_encoded], axes=(2, 2))
match = Activation('softmax')(match)

# 将匹配矩阵与第二个输入向量序列相加
response = add([match, input_encoded_c]) # (samples, story_maxlen,
query_maxlen)
response = Permute((2, 1))(response) # (samples, query_maxlen, story_maxlen)

# 拼接匹配矩阵和问题向量序列
answer = concatenate([response, question_encoded])

# 原始论文使用一个矩阵乘法来进行归约操作。
# 我们在此选择使用 RNN。
answer = LSTM(32)(answer) # (samples, 32)

# 一个正则化层 - 可能还需要更多层
answer = Dropout(0.3)(answer)
answer = Dense(vocab_size)(answer) # (samples, vocab_size)
# 输出词汇表的一个概率分布
answer = Activation('softmax')(answer)

# 构建最终模型
model = Model([input_sequence, question], answer)

```

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

# 训练
model.fit([inputs_train, queries_train], answers_train,
          batch_size=32,
          epochs=120,
          validation_data=([inputs_test, queries_test], answers_test))
```



# 在 CIFAR10 小型图像数据集上训练一个深度卷积神经网络。

在 25 轮迭代后 验证集准确率达到 75%，在 50 轮后达到 79%。 (尽管目前仍然欠拟合)。

```
from __future__ import print_function
import keras
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
import os

batch_size = 32
num_classes = 10
epochs = 100
data_augmentation = True
num_predictions = 20
save_dir = os.path.join(os.getcwd(), 'saved_models')
model_name = 'keras_cifar10_trained_model.h5'

# 数据，切分为训练和测试集。
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# 将类向量转换为二进制矩阵。
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

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

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
```

```
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))

# 初始化 RMSprop 优化器。
opt = keras.optimizers.RMSprop(learning_rate=0.0001, decay=1e-6)

# 利用 RMSprop 来训练模型。
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(x_train, y_train,
               batch_size=batch_size,
               epochs=epochs,
               validation_data=(x_test, y_test),
               shuffle=True)
else:
    print('Using real-time data augmentation.')
    # 这一步将进行数据处理和实时数据增益。data augmentation:
    datagen = ImageDataGenerator(
        featurewise_center=False, # 将整个数据集的均值设为0
        samplewise_center=False, # 将每个样本的均值设为0
        featurewise_std_normalization=False, # 将输入除以整个数据集的标准差
        samplewise_std_normalization=False, # 将输入除以其标准差
        zca_whitening=False, # 运用 ZCA 白化
        zca_epsilon=1e-06, # ZCA 白化的 epsilon值
        rotation_range=0, # 随机旋转图像范围 (角度, 0 to 180)
        # 随机水平移动图像 (总宽度的百分比)
        width_shift_range=0.1,
        # 随机垂直移动图像 (总高度的百分比)
        height_shift_range=0.1,
        shear_range=0., # 设置随机裁剪范围
        zoom_range=0., # 设置随机放大范围
        channel_shift_range=0., # 设置随机通道切换的范围
        # 设置填充输入边界之外的点的模式
        fill_mode='nearest',
        cval=0., # 在 fill_mode = "constant" 时使用的值
        horizontal_flip=True, # 随机水平翻转图像
        vertical_flip=False, # 随机垂直翻转图像
        # 设置缩放因子 (在其他转换之前使用)
        rescale=None,
        # 设置将应用于每一个输入的函数
        preprocessing_function=None,
        # 图像数据格式, "channels_first" 或 "channels_last" 之一
        data_format=None,
        # 保留用于验证的图像比例 (严格在0和1之间)
        validation_split=0.0)

    # 计算特征标准化所需的计算量
```

```
# (如果应用 ZCA 白化, 则为 std, mean和主成分).
datagen.fit(x_train)

# 利用由 datagen.flow() 生成的批来训练模型
model.fit_generator(datagen.flow(x_train, y_train,
                                 batch_size=batch_size),
                     epochs=epochs,
                     validation_data=(x_test, y_test),
                     workers=4)

# 保存模型和权重
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
model_path = os.path.join(save_dir, model_name)
model.save(model_path)
print('Saved trained model at %s' % model_path)

# 评估训练模型
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```



# 在 CIFAR10 数据集上训练 ResNet。

ResNet v1: Deep Residual Learning for Image Recognition

ResNet v2: Identity Mappings in Deep Residual Networks

Model	n	200-epoch accuracy	Original paper accuracy	sec/epoch GTX1080Ti
ResNet20 v1	3	92.16 %	91.25 %	35
ResNet32 v1	5	92.46 %	92.49 %	50
ResNet44 v1	7	92.50 %	92.83 %	70
ResNet56 v1	9	92.71 %	93.03 %	90
ResNet110 v1	18	92.65 %	93.39+-16 %	165
ResNet164 v1	27	- %	94.07 %	-
ResNet1001 v1	N/A	- %	92.39 %	-

Model	n	200-epoch accuracy	Original paper accuracy	sec/epoch GTX1080Ti
ResNet20 v2	2	- %	- %	-
ResNet32 v2	N/A	NA %	NA %	NA
ResNet44 v2	N/A	NA %	NA %	NA
ResNet56 v2	6	93.01 %	NA %	100

Model	n	200-epoch accuracy	Original paper accuracy	sec/epoch GTX1080Ti
ResNet110 v2	12	93.15 %	93.63 %	180
ResNet164 v2	18	- %	94.54 %	-
ResNet1001 v2	111	- %	95.08+-14 %	-

```

from __future__ import print_function
import keras
from keras.layers import Dense, Conv2D, BatchNormalization, Activation
from keras.layers import AveragePooling2D, Input, Flatten
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint, LearningRateScheduler
from keras.callbacks import ReduceLROnPlateau
from keras.preprocessing.image import ImageDataGenerator
from keras.regularizers import l2
from keras import backend as K
from keras.models import Model
from keras.datasets import cifar10
import numpy as np
import os

# 训练参数
batch_size = 32 # 原论文按照 batch_size=128 训练所有的网络
epochs = 200
data_augmentation = True
num_classes = 10

# 减去像素均值可提高准确度
subtract_pixel_mean = True

# 模型参数
# -----
# Model | n | 200-epoch | Orig Paper| 200-epoch | Orig Paper| sec/epoch
#       |v1(v2)| %Accuracy | %Accuracy | %Accuracy | %Accuracy | v1 (v2)
# -----
# ResNet20 | 3 (2) | 92.16 | 91.25 | ----- | ----- | 35 (---)
# ResNet32 | 5(NA) | 92.46 | 92.49 | NA | NA | 50 ( NA)
# ResNet44 | 7(NA) | 92.50 | 92.83 | NA | NA | 70 ( NA)
# ResNet56 | 9 (6) | 92.71 | 93.03 | 93.01 | NA | 90 (100)
# ResNet110 |18(12)| 92.65 | 93.39+-16| 93.15 | 93.63 | 165(180)
# ResNet164 |27(18)| ----- | 94.07 | ----- | 94.54 | ---(---)
# ResNet1001| (111)| ----- | 92.39 | ----- | 95.08+-14 | ---(---)
# -----
n = 3

# 模型版本

```

```

# Orig paper: version = 1 (ResNet v1), Improved ResNet: version = 2 (ResNet
v2)
version = 1

# 从提供的模型参数 n 计算的深度
if version == 1:
    depth = n * 6 + 2
elif version == 2:
    depth = n * 9 + 2

# 模型名称、深度和版本
model_type = 'ResNet%dv%d' % (depth, version)

# 载入 CIFAR10 数据。
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# 输入图像维度。
input_shape = x_train.shape[1:]

# 数据标准化。
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255

# 如果使用减去像素均值
if subtract_pixel_mean:
    x_train_mean = np.mean(x_train, axis=0)
    x_train -= x_train_mean
    x_test -= x_train_mean

print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print('y_train shape:', y_train.shape)

# 将类向量转换为二进制类矩阵。
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

def lr_schedule(epoch):
    """学习率调度

    学习率将在 80, 120, 160, 180 轮后依次下降。
    他作为训练期间回调的一部分，在每个时期自动调用。

    # 参数
    epoch (int): 轮次

    # 返回
    lr (float32): 学习率
    """
    lr = 1e-3
    if epoch > 180:
        lr *= 0.5e-3
    elif epoch > 160:
        lr *= 1e-3
    elif epoch > 120:

```

```

        lr *= 1e-2
    elif epoch > 80:
        lr *= 1e-1
    print('Learning rate: ', lr)
    return lr

def resnet_layer(inputs,
                 num_filters=16,
                 kernel_size=3,
                 strides=1,
                 activation='relu',
                 batch_normalization=True,
                 conv_first=True):
    """2D 卷积批量标准化 - 激活栈构建器

    # 参数
        inputs (tensor): 从输入图像或前一层来的输入张量
        num_filters (int): Conv2D 过滤器数量
        kernel_size (int): Conv2D 方形核维度
        strides (int): Conv2D 方形步幅维度
        activation (string): 激活函数名
        batch_normalization (bool): 是否包含批标准化
        conv_first (bool): conv-bn-activation (True) 或
                           bn-activation-conv (False)

    # 返回
        x (tensor): 作为下一层输入的张量
    """
    conv = Conv2D(num_filters,
                  kernel_size=kernel_size,
                  strides=strides,
                  padding='same',
                  kernel_initializer='he_normal',
                  kernel_regularizer=l2(1e-4))

    x = inputs
    if conv_first:
        x = conv(x)
        if batch_normalization:
            x = BatchNormalization()(x)
        if activation is not None:
            x = Activation(activation)(x)
    else:
        if batch_normalization:
            x = BatchNormalization()(x)
        if activation is not None:
            x = Activation(activation)(x)
        x = conv(x)
    return x

def resnet_v1(input_shape, depth, num_classes=10):
    """ResNet 版本 1 模型构建器 [a]

    2 x (3 x 3) Conv2D-BN-ReLU 的堆栈
    最后一个 ReLU 在快捷连接之后。
    """

```

在每个阶段的开始，特征图大小由具有 `strides=2` 的卷积层减半（下采样），而滤波器的数量加倍。在每个阶段中，这些层具有相同数量的过滤器和相同的特征图尺寸。

特征图尺寸：

```
stage 0: 32x32, 16
stage 1: 16x16, 32
stage 2: 8x8, 64
参数数量与 [a] 中表 6 接近：
ResNet20 0.27M
ResNet32 0.46M
ResNet44 0.66M
ResNet56 0.85M
ResNet110 1.7M
```

```
# 参数
    input_shape (tensor): 输入图像张量的尺寸
    depth (int): 核心卷积层的数量
    num_classes (int): 类别数 (CIFAR10 为 10)

# 返回
    model (Model): Keras 模型实例
"""
if (depth - 2) % 6 != 0:
    raise ValueError('depth should be 6n+2 (eg 20, 32, 44 in [a])')
# 开始模型定义
num_filters = 16
num_res_blocks = int((depth - 2) / 6)

inputs = Input(shape=input_shape)
x = resnet_layer(inputs=inputs)
# 实例化残差单元的堆栈
for stack in range(3):
    for res_block in range(num_res_blocks):
        strides = 1
        if stack > 0 and res_block == 0: # 第一层但不是第一个栈
            strides = 2 # downsample
        y = resnet_layer(inputs=x,
                          num_filters=num_filters,
                          strides=strides)
        y = resnet_layer(inputs=y,
                          num_filters=num_filters,
                          activation=None)
        if stack > 0 and res_block == 0: # first layer but not first
            stack
            # 线性投影残差快捷键连接，以匹配更改的 dims
            x = resnet_layer(inputs=x,
                              num_filters=num_filters,
                              kernel_size=1,
                              strides=strides,
                              activation=None,
                              batch_normalization=False)
            x = keras.layers.add([x, y])
            x = Activation('relu')(x)
            num_filters *= 2

        # 在顶层加分类器。
        # v1 不在最后一个快捷连接 ReLU 后使用 BN
        x = AveragePooling2D(pool_size=8)(x)
```

```

y = Flatten()(x)
outputs = Dense(num_classes,
                 activation='softmax',
                 kernel_initializer='he_normal')(y)

# 实例化模型。
model = Model(inputs=inputs, outputs=outputs)
return model


def resnet_v2(input_shape, depth, num_classes=10):
    """ResNet 版本 2 模型构建器 [b]

    (1 x 1)-(3 x 3)-(1 x 1) BN-ReLU-Conv2D 的堆栈，也被称为瓶颈层。
    每一层的第一个快捷连接是一个 1 x 1 Conv2D。
    第二个及以后的快捷连接是 identity。
    在每个阶段的开始，特征图大小由具有 strides=2 的卷积层减半（下采样），
    而滤波器的数量加倍。在每个阶段中，这些层具有相同数量的过滤器和相同的特征图尺寸。
    特征图尺寸：
    conv1 : 32x32, 16
    stage 0: 32x32, 64
    stage 1: 16x16, 128
    stage 2: 8x8, 256

    # 参数
    input_shape (tensor): 输入图像张量的尺寸
    depth (int): 核心卷积层的数量
    num_classes (int): 类别数 (CIFAR10 为 10)

    # 返回
    model (Model): Keras 模型实例
    """
    if (depth - 2) % 9 != 0:
        raise ValueError('depth should be 9n+2 (eg 56 or 110 in [b])')
    # 开始模型定义。
    num_filters_in = 16
    num_res_blocks = int((depth - 2) / 9)

    inputs = Input(shape=input_shape)
    # v2 在将输入分离为两个路径前执行带 BN-ReLU 的 Conv2D 操作。
    x = resnet_layer(inputs=inputs,
                      num_filters=num_filters_in,
                      conv_first=True)

    # 实例化残差单元的栈
    for stage in range(3):
        for res_block in range(num_res_blocks):
            activation = 'relu'
            batch_normalization = True
            strides = 1
            if stage == 0:
                num_filters_out = num_filters_in * 4
                if res_block == 0: # first layer and first stage
                    activation = None
                    batch_normalization = False
            else:
                num_filters_out = num_filters_in * 2

```

```

        if res_block == 0: # first layer but not first stage
            strides = 2 # downsample

        # 瓶颈残差单元
        y = resnet_layer(inputs=x,
                          num_filters=num_filters_in,
                          kernel_size=1,
                          strides=strides,
                          activation=activation,
                          batch_normalization=batch_normalization,
                          conv_first=False)
        y = resnet_layer(inputs=y,
                          num_filters=num_filters_in,
                          conv_first=False)
        y = resnet_layer(inputs=y,
                          num_filters=num_filters_out,
                          kernel_size=1,
                          conv_first=False)
        if res_block == 0:
            # 线性投影残差快捷键连接, 以匹配更改的 dims
            x = resnet_layer(inputs=x,
                              num_filters=num_filters_out,
                              kernel_size=1,
                              strides=strides,
                              activation=None,
                              batch_normalization=False)
        x = keras.layers.add([x, y])

    num_filters_in = num_filters_out

    # 在顶层添加分类器
    # v2 has BN-ReLU before Pooling
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = AveragePooling2D(pool_size=8)(x)
    y = Flatten()(x)
    outputs = Dense(num_classes,
                   activation='softmax',
                   kernel_initializer='he_normal')(y)

    # 实例化模型。
    model = Model(inputs=inputs, outputs=outputs)
    return model

if version == 2:
    model = resnet_v2(input_shape=input_shape, depth=depth)
else:
    model = resnet_v1(input_shape=input_shape, depth=depth)

model.compile(loss='categorical_crossentropy',
              optimizer=Adam(lr=lr_schedule(0)),
              metrics=['accuracy'])
model.summary()
print(model_type)

# 准备模型保存路径。

```

```
save_dir = os.path.join(os.getcwd(), 'saved_models')
model_name = 'cifar10_%s_model.{epoch:03d}.h5' % model_type
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
filepath = os.path.join(save_dir, model_name)

# 准备保存模型和学习速率调整的回调。
checkpoint = ModelCheckpoint(filepath=filepath,
                             monitor='val_acc',
                             verbose=1,
                             save_best_only=True)

lr_scheduler = LearningRateScheduler(lr_schedule)

lr_reducer = ReduceLROnPlateau(factor=np.sqrt(0.1),
                               cooldown=0,
                               patience=5,
                               min_lr=0.5e-6)

callbacks = [checkpoint, lr_reducer, lr_scheduler]

# 运行训练，是否数据增强可选。
if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              validation_data=(x_test, y_test),
              shuffle=True,
              callbacks=callbacks)
else:
    print('Using real-time data augmentation.')
    # 这将做预处理和实时数据增强。
    datagen = ImageDataGenerator(
        # 在整个数据集上将输入均值置为 0
        featurewise_center=False,
        # 将每个样本均值置为 0
        samplewise_center=False,
        # 将输入除以整个数据集的 std
        featurewise_std_normalization=False,
        # 将每个输入除以其自身 std
        samplewise_std_normalization=False,
        # 应用 ZCA 白化
        zca_whitening=False,
        # ZCA 白化的 epsilon 值
        zca_epsilon=1e-06,
        # 随机图像旋转角度范围 (deg 0 to 180)
        rotation_range=0,
        # 随机水平平移图像
        width_shift_range=0.1,
        # 随机垂直平移图像
        height_shift_range=0.1,
        # 设置随机裁剪范围
        shear_range=0.,
        # 设置随机缩放范围
        zoom_range=0.,
        # 设置随机通道切换范围
```

```
channel_shift_range=0.,
# 设置输入边界之外的点的数据填充模式
fill_mode='nearest',
# 在 fill_mode = "constant" 时使用的值
cval=0.,
# 随机翻转图像
horizontal_flip=True,
# 随机翻转图像
vertical_flip=False,
# 设置重缩放因子 (应用在其他任何变换之前)
rescale=None,
# 设置应用在每一个输入的预处理函数
preprocessing_function=None,
# 图像数据格式 "channels_first" 或 "channels_last" 之一
data_format=None,
# 保留用于验证的图像的比例 (严格控制在 0 和 1 之间)
validation_split=0.0)

# 计算大量的特征标准化操作
# (如果应用 ZCA 白化, 则计算 std, mean, 和 principal components)。
datagen.fit(x_train)

# 在由 datagen.flow() 生成的批次上拟合模型。
model.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_size),
                     validation_data=(x_test, y_test),
                     epochs=epochs, verbose=1, workers=4,
                     callbacks=callbacks)

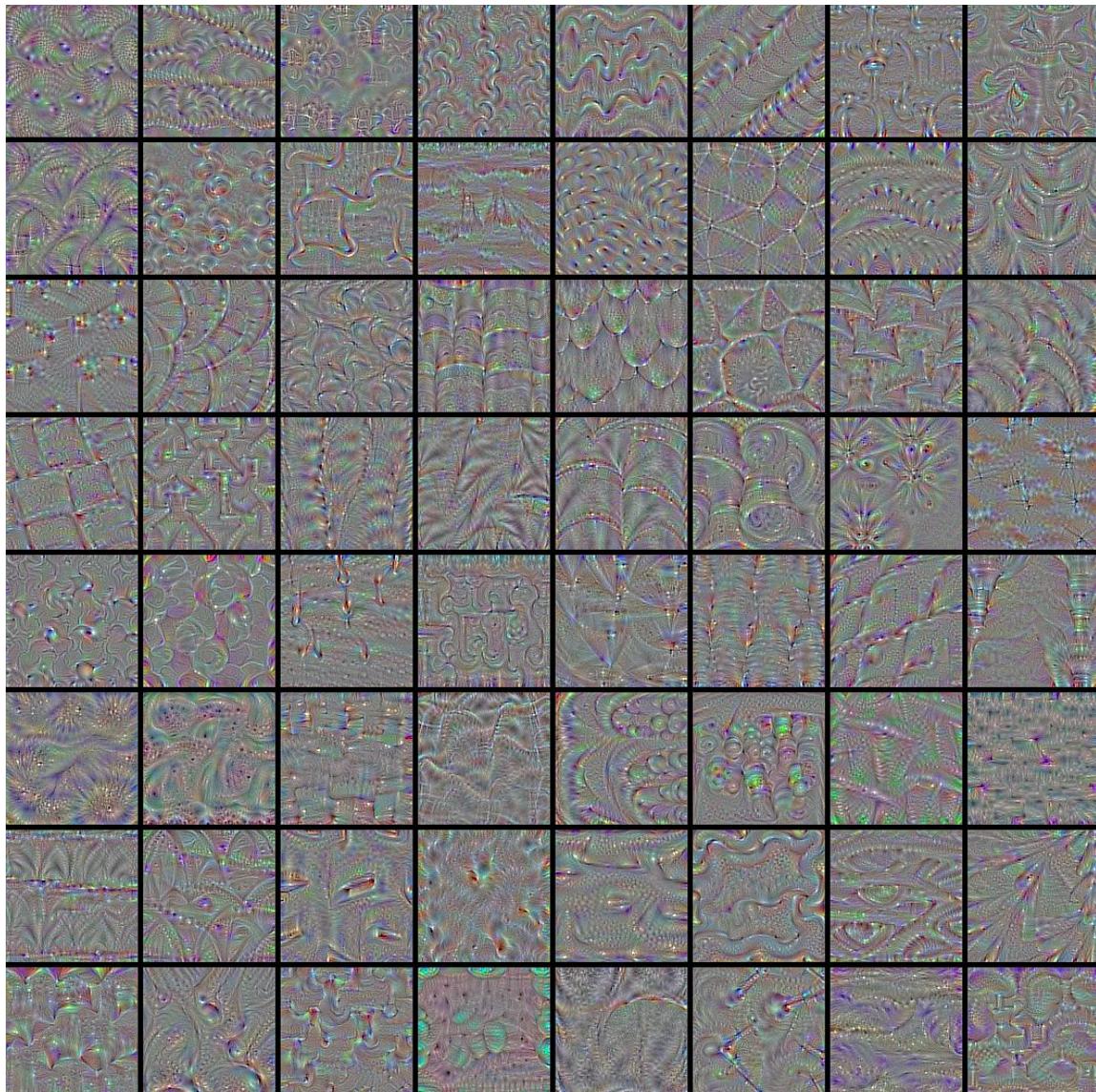
# 评估训练模型
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```



# 可视化 VGG16 的过滤器，通过输入空间梯度提升。

该脚本可以在几分钟内在 CPU 上运行完。

结果示例:



```
from __future__ import print_function

import time
import numpy as np
from PIL import Image as pil_image
from keras.preprocessing.image import save_img
from keras import layers
from keras.applications import vgg16
```

```
from keras import backend as K

def normalize(x):
    """用于标准化张量的实用函数。

    # 参数
    x: 输入张量。

    # 返回
    标准化的输入张量。
    """
    return x / (K.sqrt(K.mean(K.square(x))) + K.epsilon())

def deprocess_image(x):
    """用于将 float 数组转换为有效 uint8 图像的实用函数。

    # 参数
    x: 表示生成图像的 numpy 数组。

    # 返回
    经处理的 numpy 阵列，可用于 imshow 等。
    """
    # 标准化张量：center 为 0.，保证 std 为 0.25
    x -= x.mean()
    x /= (x.std() + K.epsilon())
    x *= 0.25

    # 裁剪为 [0, 1]
    x += 0.5
    x = np.clip(x, 0, 1)

    # 转换为 RGB 数组
    x *= 255
    if K.image_data_format() == 'channels_first':
        x = x.transpose((1, 2, 0))
    x = np.clip(x, 0, 255).astype('uint8')
    return x

def process_image(x, former):
    """用于将 float 数组转换为有效 uint8 图像转换回 float 数组的实用函数。
    `deprocess_image` 反向操作。

    # 参数
    x: numpy 数组，可用于 imshow 等。
    former: 前身 numpy 数组，
            需要确定前者的均值和方差。

    # 返回
    一个处理后的 numpy 数组，表示一幅生成图像。
    """
    if K.image_data_format() == 'channels_first':
        x = x.transpose((2, 0, 1))
    return (x / 255 - 0.5) * 4 * former.std() + former.mean()
```

```

def visualize_layer(model,
                    layer_name,
                    step=1.,
                    epochs=15,
                    upscaling_steps=9,
                    upscaling_factor=1.2,
                    output_dim=(412, 412),
                    filter_range=(0, None)):
    """可视化某个模型中一个转换层的最相关过滤器。

    # 参数
    model: 包含 layer_name 的模型。
    layer_name: 需要可视化的层的名称。
        必须是模型的一部分。
    step: 梯度提升步长。
    epochs: 梯度提升迭代轮次。
    upscaling_steps: upscaling 步数。
        起始图像为 (80, 80)。
    upscaling_factor: 将图像缓慢提升到 output_dim 的因子。
    output_dim: [img_width, img_height] 输出图像维度。
    filter_range: 元组 [lower, upper]
        决定需要计算的过滤器数目。
        如果第二个值为 `None`,
        最后一个过滤器将被推断为上边界。
    """

    def _generate_filter_image(input_img,
                               layer_output,
                               filter_index):
        """为一个特定的过滤器生成图像。

        # 参数
        input_img: 输入图像张量。
        layer_output: 输出图像张量。
        filter_index: 需要处理的过滤器数目。
            假设可用。

        # 返回
        要么是 None, 如果无法生成图像。
        要么是图像 (数组) 本身以及最后的 loss 组成的元组。
        """

        s_time = time.time()

        # 构建一个损失函数, 使所考虑的层的第 n 个过滤器的激活最大化
        if K.image_data_format() == 'channels_first':
            loss = K.mean(layer_output[:, filter_index, :, :])
        else:
            loss = K.mean(layer_output[:, :, :, filter_index])

        # 计算这种损失的输入图像的梯度
        grads = K.gradients(loss, input_img)[0]

        # 标准化技巧: 将梯度标准化
        grads = normalize(grads)

        # 此函数返回给定输入图片的损失和梯度

```

```

iterate = K.function([input_img], [loss, grads])

# 从带有一些随机噪音的灰色图像开始
intermediate_dim = tuple(
    int(x / (upsampling_factor ** upscaling_steps)) for x in
output_dim)
if K.image_data_format() == 'channels_first':
    input_img_data = np.random.random(
        (1, 3, intermediate_dim[0], intermediate_dim[1])))
else:
    input_img_data = np.random.random(
        (1, intermediate_dim[0], intermediate_dim[1], 3)))
input_img_data = (input_img_data - 0.5) * 20 + 128

# 缓慢放大原始图像的尺寸可以防止可视化结构的主导高频现象发生
# (如果我们直接计算 412d-image 时该现象就会发生。)
# 作为每个后续维度的更好起点, 因此它避免了较差的局部最小值
for up in reversed(range(upscaling_steps)):
    # 执行 20 次梯度提升
    for _ in range(epochs):
        loss_value, grads_value = iterate([input_img_data])
        input_img_data += grads_value * step

    # 一些过滤器被卡在了 0, 我们可以跳过它们
    if loss_value <= K.epsilon():
        return None

    # 计算放大维度
    intermediate_dim = tuple(
        int(x / (upsampling_factor ** up)) for x in output_dim)
    # 放大
    img = deprocess_image(input_img_data[0])
    img = np.array(pil_image.fromarray(img).resize(intermediate_dim,
                                                   pil_image.BICUBIC))
    input_img_data = np.expand_dims(
        process_image(img, input_img_data[0]), 0)

    # 解码生成的输入图像
    img = deprocess_image(input_img_data[0])
    e_time = time.time()
    print('Costs of filter {:3}: {:.5f} ( {:.2f}
s )'.format(filter_index,
               loss_value,
               e_time -
s_time))
    return img, loss_value

def _draw_filters(filters, n=None):
    """在 nxn 网格中绘制最佳过滤器。

    # 参数
        filters: 每个已处理过滤器的生成图像及其相应的损失的列表。
        n: 网格维度。
            如果为 None, 将使用最大可能的方格
    """
    if n is None:
        n = int(np.floor(np.sqrt(len(filters))))

```

```

# 假设损失最大的过滤器看起来更好看。
# 我们只保留顶部 n*n 过滤器。
filters.sort(key=lambda x: x[1], reverse=True)
filters = filters[:n * n]

# 构建一个有足够空间的黑色图像
# 例如, 8 x 8 个过滤器, 总尺寸为 412 x 412, 每个过滤器 5px 间隔的图像
MARGIN = 5
width = n * output_dim[0] + (n - 1) * MARGIN
height = n * output_dim[1] + (n - 1) * MARGIN
stitched_filters = np.zeros((width, height, 3), dtype='uint8')

# 用我们保存的过滤器填充图像
for i in range(n):
    for j in range(n):
        img, _ = filters[i * n + j]
        width_margin = (output_dim[0] + MARGIN) * i
        height_margin = (output_dim[1] + MARGIN) * j
        stitched_filters[
            width_margin: width_margin + output_dim[0],
            height_margin: height_margin + output_dim[1], :] = img

# 将结果保存到磁盘
save_img('vgg_{0:}_{1:}x{1:}.png'.format(layer_name, n),
stitched_filters)

# 这是输入图像的占位符
assert len(model.inputs) == 1
input_img = model.inputs[0]

# 获取每个『关键』图层的符号输出 (我们给它们唯一的名称)。
layer_dict = dict([(layer.name, layer) for layer in model.layers[1:]])

output_layer = layer_dict[layer_name]
assert isinstance(output_layer, layers.Conv2D)

# 计算要处理的过滤范围
filter_lower = filter_range[0]
filter_upper = (filter_range[1]
                if filter_range[1] is not None
                else len(output_layer.get_weights()[1]))
assert(filter_lower >= 0
       and filter_upper <= len(output_layer.get_weights()[1])
       and filter_upper > filter_lower)
print('Compute filters {} to {}'.format(filter_lower, filter_upper))

# 迭代每个过滤器并生成其相应的图像
processed_filters = []
for f in range(filter_lower, filter_upper):
    img_loss = _generate_filter_image(input_img, output_layer.output, f)

    if img_loss is not None:
        processed_filters.append(img_loss)

print('{} filter processed.'.format(len(processed_filters)))
# Finally draw and store the best filters to disk

```

```
_draw_filters(processed_filters)

if __name__ == '__main__':
    # 我们想要可视化的图层的名称
    # (see model definition at keras/applications/vgg16.py)
    LAYER_NAME = 'block5_conv1'

    # 构建 ImageNet 权重预训练的 VGG16 网络
    vgg = vgg16.VGG16(weights='imagenet', include_top=False)
    print('Model loaded.')
    vgg.summary()

    # 调用示例函数
    visualize_layer(vgg, LAYER_NAME)
```



# 此脚本演示了卷积LSTM网络的使用。

该网络用于预测包含移动方块的人工生成的电影的下一帧。

```
from keras.models import Sequential
from keras.layers.convolutional import Conv3D
from keras.layers.convolutional_recurrent import ConvLSTM2D
from keras.layers.normalization import BatchNormalization
import numpy as np
import pylab as plt

# 我们创建一个网络层，以尺寸为 (n_frames, width, height, channels) 的电影作为输入，并返回相同尺寸的电影。
seq = Sequential()
seq.add(ConvLSTM2D(filters=40, kernel_size=(3, 3),
                   input_shape=(None, 40, 40, 1),
                   padding='same', return_sequences=True))
seq.add(BatchNormalization())

seq.add(ConvLSTM2D(filters=40, kernel_size=(3, 3),
                   padding='same', return_sequences=True))
seq.add(BatchNormalization())

seq.add(ConvLSTM2D(filters=40, kernel_size=(3, 3),
                   padding='same', return_sequences=True))
seq.add(BatchNormalization())

seq.add(ConvLSTM2D(filters=40, kernel_size=(3, 3),
                   padding='same', return_sequences=True))
seq.add(BatchNormalization())

seq.add(Conv3D(filters=1, kernel_size=(3, 3, 3),
              activation='sigmoid',
              padding='same', data_format='channels_last'))
seq.compile(loss='binary_crossentropy', optimizer='adadelta')

# 人工数据生成：
# 生成内部有3到7个移动方块的电影。
# 方块的尺寸为 1x1 或 2x2 像素，
# 随着时间的推移线性移动。
# 为方便起见，我们首先创建宽度和高度较大的电影 (80x80)，最后选择 40x40 的窗口。

def generate_movies(n_samples=1200, n_frames=15):
    row = 80
    col = 80
    noisy_movies = np.zeros((n_samples, n_frames, row, col, 1),
                           dtype=np.float)
    shifted_movies = np.zeros((n_samples, n_frames, row, col, 1),
                             dtype=np.float)

    for i in range(n_samples):
        # 添加 3 到 7 个移动方块
```

```

n = np.random.randint(3, 8)

for j in range(n):
    # 初始位置
    xstart = np.random.randint(20, 60)
    ystart = np.random.randint(20, 60)
    # 运动方向
    directionx = np.random.randint(0, 3) - 1
    directiony = np.random.randint(0, 3) - 1

    # 方块尺寸
    w = np.random.randint(2, 4)

    for t in range(n_frames):
        x_shift = xstart + directionx * t
        y_shift = ystart + directiony * t
        noisy_movies[i, t, x_shift - w: x_shift + w,
                    y_shift - w: y_shift + w, 0] += 1

    # 通过添加噪音使其更加健壮。
    # 这个想法是，如果在推理期间，像素的值不是一个，
    # 我们需要训练更加健壮的网络，并仍然将其视为属于方块的像素。
    if np.random.randint(0, 2):
        noise_f = (-1)**np.random.randint(0, 2)
        noisy_movies[i, t,
                    x_shift - w - 1: x_shift + w + 1,
                    y_shift - w - 1: y_shift + w + 1,
                    0] += noise_f * 0.1

    # Shift the ground truth by 1
    x_shift = xstart + directionx * (t + 1)
    y_shift = ystart + directiony * (t + 1)
    shifted_movies[i, t, x_shift - w: x_shift + w,
                  y_shift - w: y_shift + w, 0] += 1

# 裁剪为 40x40 窗口
noisy_movies = noisy_movies[:, :, 20:60, 20:60, :]
shifted_movies = shifted_movies[:, :, 20:60, 20:60, :]
noisy_movies[noisy_movies >= 1] = 1
shifted_movies[shifted_movies >= 1] = 1
return noisy_movies, shifted_movies

# 训练网络
noisy_movies, shifted_movies = generate_movies(n_samples=1200)
seq.fit(noisy_movies[:1000], shifted_movies[:1000], batch_size=10,
        epochs=300, validation_split=0.05)

# 在一部电影上测试网络
# 用前 7 个位置训练它，然后预测新的位置
which = 1004
track = noisy_movies[which][:7, :, :, :]

for j in range(16):
    new_pos = seq.predict(track[np.newaxis, :, :, :, :, :])
    new = new_pos[:, -1, :, :, :]
    track = np.concatenate((track, new), axis=0)

```

```
# 然后将预测与实际进行比较
track2 = noisy_movies[which][::, ::, ::, ::]
for i in range(15):
    fig = plt.figure(figsize=(10, 5))

    ax = fig.add_subplot(121)

    if i >= 7:
        ax.text(1, 3, 'Predictions !', fontsize=20, color='w')
    else:
        ax.text(1, 3, 'Initial trajectory', fontsize=20)

    toplot = track[i, ::, ::, 0]

    plt.imshow(toplot)
    ax = fig.add_subplot(122)
    plt.text(1, 3, 'Ground truth', fontsize=20)

    toplot = track2[i, ::, ::, 0]
    if i >= 2:
        toplot = shifted_movies[which][i - 1, ::, ::, 0]

    plt.imshow(toplot)
    plt.savefig('%i_animate.png' % (i + 1))
```



# Keras 实现的 Deep Dreaming。

按以下命令执行该脚本：

```
python deep_dream.py path_to_your_base_image.jpg prefix_for_results
```

例如：

```
python deep_dream.py img/mypic.jpg results/dream
```

```
from __future__ import print_function

from keras.preprocessing.image import load_img, save_img, img_to_array
import numpy as np
import scipy
import argparse

from keras.applications import inception_v3
from keras import backend as K

parser = argparse.ArgumentParser(description='Deep Dreams with Keras.')
parser.add_argument('base_image_path', metavar='base', type=str,
                    help='Path to the image to transform.')
parser.add_argument('result_prefix', metavar='res_prefix', type=str,
                    help='Prefix for the saved results.')

args = parser.parse_args()
base_image_path = args.base_image_path
result_prefix = args.result_prefix

# 这些是我们尝试最大化激活的层的名称，以及它们们在我们试图最大化的最终损失中的权重。
# 你可以调整这些设置以获得新的视觉效果。
settings = {
    'features': {
        'mixed2': 0.2,
        'mixed3': 0.5,
        'mixed4': 2.,
        'mixed5': 1.5,
    },
}

def preprocess_image(image_path):
    # 用于打开，调整图片大小并将图片格式化为适当的张量的实用函数。
    img = load_img(image_path)
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = inception_v3.preprocess_input(img)
    return img
```

```

def deprocess_image(x):
    # 函数将张量转换为有效图像的实用函数。
    if K.image_data_format() == 'channels_first':
        x = x.reshape((3, x.shape[2], x.shape[3]))
        x = x.transpose((1, 2, 0))
    else:
        x = x.reshape((x.shape[1], x.shape[2], 3))
    x /= 2.
    x += 0.5
    x *= 255.
    x = np.clip(x, 0, 255).astype('uint8')
    return x

K.set_learning_phase(0)

# 使用我们的占位符构建 InceptionV3 网络。
# 该模型将加载预先训练的 ImageNet 权重。
model = inception_v3.InceptionV3(weights='imagenet',
                                   include_top=False)
dream = model.input
print('Model loaded.')

# 获取每个『关键』层的符号输出（我们为它们指定了唯一的名称）。
layer_dict = dict([(layer.name, layer) for layer in model.layers])

# 定义损失。
loss = K.variable(0.)
for layer_name in settings['features']:
    # 将层特征的 L2 范数添加到损失中。
    if layer_name not in layer_dict:
        raise ValueError('Layer ' + layer_name + ' not found in model.')
    coeff = settings['features'][layer_name]
    x = layer_dict[layer_name].output
    # 我们通过仅涉及损失中的非边界像素来避免边界伪影。
    scaling = K.prod(K.cast(K.shape(x), 'float32'))
    if K.image_data_format() == 'channels_first':
        loss = loss + coeff * K.sum(K.square(x[:, :, 2:-2, 2:-2])) / scaling
    else:
        loss = loss + coeff * K.sum(K.square(x[:, 2:-2, 2:-2, :])) / scaling

# 计算 dream 即损失的梯度。
grads = K.gradients(loss, dream)[0]
# 标准化梯度。
grads /= K.maximum(K.mean(K.abs(grads)), K.epsilon())

# 设置函数，以检索给定输入图像的损失和梯度的值。
outputs = [loss, grads]
fetch_loss_and_grads = K.function([dream], outputs)

def eval_loss_and_grads(x):
    outs = fetch_loss_and_grads([x])
    loss_value = outs[0]
    grad_values = outs[1]
    return loss_value, grad_values

```

```

def resize_img(img, size):
    img = np.copy(img)
    if K.image_data_format() == 'channels_first':
        factors = (1, 1,
                    float(size[0]) / img.shape[2],
                    float(size[1]) / img.shape[3])
    else:
        factors = (1,
                    float(size[0]) / img.shape[1],
                    float(size[1]) / img.shape[2],
                    1)
    return scipy.ndimage.zoom(img, factors, order=1)

def gradient_ascent(x, iterations, step, max_loss=None):
    for i in range(iterations):
        loss_value, grad_values = eval_loss_and_grads(x)
        if max_loss is not None and loss_value > max_loss:
            break
        print('..Loss value at', i, ':', loss_value)
        x += step * grad_values
    return x

```

"""Process:

- 载入原始图像。
- 定义一系列预处理规模（即图像尺寸），从最小到最大。
- 将原始图像调整为最小尺寸。
- 对于每个规模，从最小的（即当前的）开始：
  - 执行梯度提升
  - 将图像放大到下一个比例
  - 重新投射在提升时丢失的细节
- 当我们回到原始大小时停止。

为了获得在放大过程中丢失的细节，我们只需将原始图像缩小，放大，然后将结果与（调整大小的）原始图像进行比较即可。

```

# 把玩这些超参数也可以让你获得新的效果
step = 0.01 # 梯度提升步长
num_octave = 3 # 运行梯度提升的规模数
octave_scale = 1.4 # 规模之间的比
iterations = 20 # 每个规模的提升步数
max_loss = 10.

img = preprocess_image(base_image_path)
if K.image_data_format() == 'channels_first':
    original_shape = img.shape[2:]
else:
    original_shape = img.shape[1:3]
successive_shapes = [original_shape]
for i in range(1, num_octave):
    shape = tuple([int(dim / (octave_scale ** i)) for dim in original_shape])
    successive_shapes.append(shape)
successive_shapes = successive_shapes[:-1]

```

```
original_img = np.copy(img)
shrunk_original_img = resize_img(img, successive_shapes[0])

for shape in successive_shapes:
    print('Processing image shape', shape)
    img = resize_img(img, shape)
    img = gradient_ascent(img,
                          iterations=iterations,
                          step=step,
                          max_loss=max_loss)
    upscaled_shrunk_original_img = resize_img(shrunk_original_img, shape)
    same_size_original = resize_img(original_img, shape)
    lost_detail = same_size_original - upscaled_shrunk_original_img

    img += lost_detail
    shrunk_original_img = resize_img(original_img, shape)

save_img(result_prefix + '.png', deprocess_image(np.copy(img)))
```



# 光学字符识别

此示例使用卷积堆栈，后跟递归堆栈和 CTC logloss 函数，以对生成的文本图像进行光学字符识别。我没有证据表明它实际上是在学习文本的一般形状，还是仅仅能够识别所抛出的所有不同字体……它的目的更多是为了在 Keras 中演示CTC。请注意，可能需要针对使用中的特定操作系统更新字体列表。

它从 4 个字母词开始。对于前12个轮次，使用 TextImageGenerator 类（同时是测试/训练数据的生成器类和 Keras 回调类）会逐渐增加难度。20个 轮次后，通过重新编译模型以处理更宽的图像并重建单词列表以包含两个以空格分隔的单词，将抛出更长的序列。

下表显示了标准化的编辑距离值。Theano 使用的 CTC 实现略有不同，因此结果也有所不同。

Epoch	TF	TH
10	0.027	0.064
15	0.038	0.035
20	0.043	0.045
25	0.014	0.019

## 其他依赖

需要 `cairo` 和 `editdistance` 包:

首先，安装 Cairo 库: <https://cairographics.org/>

然后安装 Python 依赖:

```
pip install cairocffi  
pip install editdistance
```

Created by Mike Henry <https://github.com/mbhenry/>

```
import os  
import itertools  
import codecs
```

```

import re
import datetime
import cairocffi as cairo
import editdistance
import numpy as np
from scipy import ndimage
import pylab
from keras import backend as K
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.layers import Input, Dense, Activation
from keras.layers import Reshape, Lambda
from keras.layers.merge import add, concatenate
from keras.models import Model
from keras.layers.recurrent import GRU
from keras.optimizers import SGD
from keras.utils.data_utils import get_file
from keras.preprocessing import image
import keras.callbacks

OUTPUT_DIR = 'image_ocr'

# 字符类和匹配的正则表达式过滤器
regex = r'^[a-z ]+$'
alphabet = u'abcdefghijklmnopqrstuvwxyz '


np.random.seed(55)

# 这会产生更大的 "斑点" 噪声,
# 看起来比仅添加高斯噪声更为真实,
# 它假定像素的灰度范围为 0 到 1

def speckle(img):
    severity = np.random.uniform(0, 0.6)
    blur = ndimage.gaussian_filter(np.random.randn(*img.shape) * severity, 1)
    img_speck = (img + blur)
    img_speck[img_speck > 1] = 1
    img_speck[img_speck <= 0] = 0
    return img_speck

# 在随机位置绘制字符串, 边界框也使用随机字体、轻微的随机旋转和随机的斑点噪声

def paint_text(text, w, h, rotate=False, ud=False, multi_fonts=False):
    surface = cairo.ImageSurface(cairo.FORMAT_RGB24, w, h)
    with cairo.Context(surface) as context:
        context.set_source_rgb(1, 1, 1) # 白色
        context.paint()
    # 此字体列表可在 CentOS 7 中使用
    if multi_fonts:
        fonts = [
            'Century Schoolbook', 'Courier', 'STIX',
            'URW Chancery L', 'FreeMono']
        context.select_font_face(
            np.random.choice(fonts),
            cairo.FONT_SLANT_NORMAL,

```

```

        np.random.choice([cairo.FONT_WEIGHT_BOLD,
cairo.FONT_WEIGHT_NORMAL]))
    else:
        context.select_font_face('Courier',
                                cairo.FONT_SLANT_NORMAL,
                                cairo.FONT_WEIGHT_BOLD)
    context.set_font_size(25)
    box = context.text_extents(text)
    border_w_h = (4, 4)
    if box[2] > (w - 2 * border_w_h[1]) or box[3] > (h - 2 *
border_w_h[0]):
        raise IOError(('Could not fit string into image.'
                      'Max char count is too large for given image
width.'))

# 通过在画布上随机放置文本框并旋转一些空间来教会 RNN 平移不变性
max_shift_x = w - box[2] - border_w_h[0]
max_shift_y = h - box[3] - border_w_h[1]
top_left_x = np.random.randint(0, int(max_shift_x))
if ud:
    top_left_y = np.random.randint(0, int(max_shift_y))
else:
    top_left_y = h // 2
context.move_to(top_left_x - int(box[0]), top_left_y - int(box[1]))
context.set_source_rgb(0, 0, 0)
context.show_text(text)

buf = surface.get_data()
a = np.frombuffer(buf, np.uint8)
a.shape = (h, w, 4)
a = a[:, :, 0] # 抓取单个通道
a = a.astype(np.float32) / 255
a = np.expand_dims(a, 0)
if rotate:
    a = image.random_rotation(a, 3 * (w - top_left_x) / w + 1)
a = speckle(a)

return a


def shuffle_mats_or_lists(matrix_list, stop_ind=None):
    ret = []
    assert all([len(i) == len(matrix_list[0]) for i in matrix_list])
    len_val = len(matrix_list[0])
    if stop_ind is None:
        stop_ind = len_val
    assert stop_ind <= len_val

    a = list(range(stop_ind))
    np.random.shuffle(a)
    a += list(range(stop_ind, len_val))
    for mat in matrix_list:
        if isinstance(mat, np.ndarray):
            ret.append(mat[a])
        elif isinstance(mat, list):
            ret.append([mat[i] for i in a])
        else:

```

```

        raise TypeError(``shuffle_mats_or_lists` only supports '
                      'numpy.array and list objects.')
    return ret

# 将字符转换为唯一的整数值
def text_to_labels(text):
    ret = []
    for char in text:
        ret.append(alphabet.find(char))
    return ret

# 将数字类反向转换回字符
def labels_to_text(labels):
    ret = []
    for c in labels:
        if c == len(alphabet): # CTC 空白
            ret.append("")
        else:
            ret.append(alphabet[c])
    return ''.join(ret)

# 仅 a-z 和空格..可能不难扩展为大写和符号
def is_valid_str(in_str):
    search = re.compile(regex, re.UNICODE).search
    return bool(search(in_str))

# 使用生成器函数提供训练/测试数据。每次使用随机扰动动态创建图像渲染和文本
class TextImageGenerator(keras.callbacks.Callback):

    def __init__(self, monogram_file, bigram_file, minibatch_size,
                 img_w, img_h, downsample_factor, val_split,
                 absolute_max_string_len=16):

        self.minibatch_size = minibatch_size
        self.img_w = img_w
        self.img_h = img_h
        self.monogram_file = monogram_file
        self.bigram_file = bigram_file
        self.downsample_factor = downsample_factor
        self.val_split = val_split
        self.blank_label = self.get_output_size() - 1
        self.absolute_max_string_len = absolute_max_string_len

    def get_output_size(self):
        return len(alphabet) + 1

    # 由于使用生成器，因此 num_words 可以与轮次大小无关，因为 max_string_len 增长，
    # num_words 也会增长
    def build_word_list(self, num_words, max_string_len=None,
                       mono_fraction=0.5):
        assert max_string_len <= self.absolute_max_string_len

```

```

        assert num_words % self.minibatch_size == 0
        assert (self.val_split * num_words) % self.minibatch_size == 0
        self.num_words = num_words
        self.string_list = [''] * self.num_words
        tmp_string_list = []
        self.max_string_len = max_string_len
        self.Y_data = np.ones([self.num_words, self.absolute_max_string_len])
* -1
        self.X_text = []
        self.Y_len = [0] * self.num_words

    def _is_length_of_word_valid(word):
        return (max_string_len == -1 or
                max_string_len is None or
                len(word) <= max_string_len)

    # 会标文件按英语语音中的频率排序
    with codecs.open(self.monogram_file, mode='r', encoding='utf-8') as f:
        for line in f:
            if len(tmp_string_list) == int(self.num_words *
mono_fraction):
                break
            word = line.rstrip()
            if _is_length_of_word_valid(word):
                tmp_string_list.append(word)

    # bigram文件包含英语语音中的常用单词对
    with codecs.open(self.bigram_file, mode='r', encoding='utf-8') as f:
        lines = f.readlines()
        for line in lines:
            if len(tmp_string_list) == self.num_words:
                break
            columns = line.lower().split()
            word = columns[0] + ' ' + columns[1]
            if is_valid_str(word) and _is_length_of_word_valid(word):
                tmp_string_list.append(word)
        if len(tmp_string_list) != self.num_words:
            raise IOError('Could not pull enough words'
                          ' from supplied monogram and bigram files.')
    # 隔行扫描以混合易用词和难用词
    self.string_list[::2] = tmp_string_list[:self.num_words // 2]
    self.string_list[1::2] = tmp_string_list[self.num_words // 2:]

    for i, word in enumerate(self.string_list):
        self.Y_len[i] = len(word)
        self.Y_data[i, 0:len(word)] = text_to_labels(word)
        self.X_text.append(word)
    self.Y_len = np.expand_dims(np.array(self.Y_len), 1)

    self.cur_val_index = self.val_split
    self.cur_train_index = 0

    # 每次从训练/验证/测试中请求图像时，都会对文本进行新的随机绘制
    def get_batch(self, index, size, train):
        # width 和 height 按典型的 Keras 约定反向，因为 width 是将其馈入 RNN 时的时间
维。
        if K.image_data_format() == 'channels_first':

```



```

        self.cur_val_index += self.minibatch_size
        if self.cur_val_index >= self.num_words:
            self.cur_val_index = self.val_split + self.cur_val_index % 32
        yield ret

    def on_train_begin(self, logs={}):
        self.build_word_list(16000, 4, 1)
        self.paint_func = lambda text: paint_text(
            text, self.img_w, self.img_h,
            rotate=False, ud=False, multi_fonts=False)

    def on_epoch_begin(self, epoch, logs={}):
        # 重新结合绘画功能以实现课程学习
        if 3 <= epoch < 6:
            self.paint_func = lambda text: paint_text(
                text, self.img_w, self.img_h,
                rotate=False, ud=True, multi_fonts=False)
        elif 6 <= epoch < 9:
            self.paint_func = lambda text: paint_text(
                text, self.img_w, self.img_h,
                rotate=False, ud=True, multi_fonts=True)
        elif epoch >= 9:
            self.paint_func = lambda text: paint_text(
                text, self.img_w, self.img_h,
                rotate=True, ud=True, multi_fonts=True)
        if epoch >= 21 and self.max_string_len < 12:
            self.build_word_list(32000, 12, 0.5)

# 尽管不是内部 Keras 损失函数，但实际损失计算仍在此处发生

    def ctc_lambda_func(args):
        y_pred, labels, input_length, label_length = args
        # 这里的 2 是至关重要的，因为 RNN 的前几个输出往往是垃圾：
        y_pred = y_pred[:, 2:, :]
        return K.ctc_batch_cost(labels, y_pred, input_length, label_length)

# 对于真正的 OCR 应用程序，这应该是带有字典和语言模型的波束搜索。
# 对于此示例，最佳路径就足够了。

    def decode_batch(test_func, word_batch):
        out = test_func([word_batch])[0]
        ret = []
        for j in range(out.shape[0]):
            out_best = list(np.argmax(out[j, 2:], 1))
            out_best = [k for k, g in itertools.groupby(out_best)]
            outstr = labels_to_text(out_best)
            ret.append(outstr)
        return ret

    class VizCallback(keras.callbacks.Callback):

        def __init__(self, run_name, test_func, text_img_gen,
                     num_display_words=6):
            self.test_func = test_func

```

```

        self.output_dir = os.path.join(
            OUTPUT_DIR, run_name)
        self.text_img_gen = text_img_gen
        self.num_display_words = num_display_words
        if not os.path.exists(self.output_dir):
            os.makedirs(self.output_dir)

    def show_edit_distance(self, num):
        num_left = num
        mean_norm_ed = 0.0
        mean_ed = 0.0
        while num_left > 0:
            word_batch = next(self.text_img_gen)[0]
            num_proc = min(word_batch['the_input'].shape[0], num_left)
            decoded_res = decode_batch(self.test_func,
                                         word_batch['the_input'][0:num_proc])
            for j in range(num_proc):
                edit_dist = editdistance.eval(decoded_res[j],
                                              word_batch['source_str'][j])
                mean_ed += float(edit_dist)
                mean_norm_ed += float(edit_dist) /
                    len(word_batch['source_str'][j])
            num_left -= num_proc
            mean_norm_ed = mean_norm_ed / num
            mean_ed = mean_ed / num
            print('\nOut of %d samples: Mean edit distance:'
                  '%.3f Mean normalized edit distance: %.3f'
                  % (num, mean_ed, mean_norm_ed))

    def on_epoch_end(self, epoch, logs={}):
        self.model.save_weights(
            os.path.join(self.output_dir, 'weights%02d.h5' % (epoch)))
        self.show_edit_distance(256)
        word_batch = next(self.text_img_gen)[0]
        res = decode_batch(self.test_func,
                           word_batch['the_input'][0:self.num_display_words])
        if word_batch['the_input'][0].shape[0] < 256:
            cols = 2
        else:
            cols = 1
        for i in range(self.num_display_words):
            pylab.subplot(self.num_display_words // cols, cols, i + 1)
            if K.image_data_format() == 'channels_first':
                the_input = word_batch['the_input'][i, 0, :, :]
            else:
                the_input = word_batch['the_input'][i, :, :, 0]
            pylab.imshow(the_input.T, cmap='Greys_r')
            pylab.xlabel(
                'Truth = \'%s\'\nDecoded = \'%s\' ' %
                (word_batch['source_str'][i], res[i]))
        fig = pylab.gcf()
        fig.set_size_inches(10, 13)
        pylab.savefig(os.path.join(self.output_dir, 'e%02d.png' % (epoch)))
        pylab.close()

    def train(run_name, start_epoch, stop_epoch, img_w):

```

```

# 输入参数
img_h = 64
words_per_epoch = 16000
val_split = 0.2
val_words = int(words_per_epoch * (val_split))

# 网络参数
conv_filters = 16
kernel_size = (3, 3)
pool_size = 2
time_dense_size = 32
rnn_size = 512
minibatch_size = 32

if K.image_data_format() == 'channels_first':
    input_shape = (1, img_w, img_h)
else:
    input_shape = (img_w, img_h, 1)

fdirectory = os.path.dirname(
    get_file('wordlists.tgz',
        origin='http://www.mythic-ai.com/datasets/wordlists.tgz',
        untar=True))

img_gen = TextImageGenerator(
    monogram_file=os.path.join(fdirectory, 'wordlist_mono_clean.txt'),
    bigram_file=os.path.join(fdirectory, 'wordlist_bi_clean.txt'),
    minibatch_size=minibatch_size,
    img_w=img_w,
    img_h=img_h,
    downsample_factor=(pool_size ** 2),
    val_split=words_per_epoch - val_words)
act = 'relu'
input_data = Input(name='the_input', shape=input_shape, dtype='float32')
inner = Conv2D(conv_filters, kernel_size, padding='same',
               activation=act, kernel_initializer='he_normal',
               name='conv1')(input_data)
inner = MaxPooling2D(pool_size=(pool_size, pool_size), name='max1')(inner)
inner = Conv2D(conv_filters, kernel_size, padding='same',
               activation=act, kernel_initializer='he_normal',
               name='conv2')(inner)
inner = MaxPooling2D(pool_size=(pool_size, pool_size), name='max2')(inner)

conv_to_rnn_dims = (img_w // (pool_size ** 2),
                   (img_h // (pool_size ** 2)) * conv_filters)
inner = Reshape(target_shape=conv_to_rnn_dims, name='reshape')(inner)

# 减少进入 RNN 的输入大小:
inner = Dense(time_dense_size, activation=act, name='dense1')(inner)

# 两层双向GRU
# 单层 GRU 似乎也可以, 如果不比 LSTM 强:
gru_1 = GRU(rnn_size, return_sequences=True,
            kernel_initializer='he_normal', name='gru1')(inner)
gru_1b = GRU(rnn_size, return_sequences=True,
             go_backwards=True, kernel_initializer='he_normal',
             name='gru1_b')(inner)

```

```

gru1_merged = add([gru_1, gru_1b])
gru_2 = GRU(rnn_size, return_sequences=True,
            kernel_initializer='he_normal', name='gru2')(gru1_merged)
gru_2b = GRU(rnn_size, return_sequences=True, go_backwards=True,
            kernel_initializer='he_normal', name='gru2_b')(gru1_merged)

# 将 RNN 输出转换为字符激活:
inner = Dense(img_gen.get_output_size(), kernel_initializer='he_normal',
              name='dense2')(concatenate([gru_2, gru_2b]))
y_pred = Activation('softmax', name='softmax')(inner)
Model(inputs=input_data, outputs=y_pred).summary()

labels = Input(name='the_labels',
               shape=[img_gen.absolute_max_string_len], dtype='float32')
input_length = Input(name='input_length', shape=[1], dtype='int64')
label_length = Input(name='label_length', shape=[1], dtype='int64')
# Keras 当前不支持带有额外参数的损失函数，因此 CTC 损失在 Lambda 层中实现
loss_out = Lambda(
    ctc_lambda_func, output_shape=(1, ),
    name='ctc')([y_pred, labels, input_length, label_length])

# clipnorm 似乎加快了收敛速度
sgd = SGD(learning_rate=0.02,
           decay=1e-6,
           momentum=0.9,
           nesterov=True)

model = Model(inputs=[input_data, labels, input_length, label_length],
               outputs=loss_out)

# 损失计算发生在其他地方，因此请使用虚拟 lambda 函数补偿损失
model.compile(loss={'ctc': lambda y_true, y_pred: y_pred}, optimizer=sgd)
if start_epoch > 0:
    weight_file = os.path.join(
        OUTPUT_DIR,
        os.path.join(run_name, 'weights%02d.h5' % (start_epoch - 1)))
    model.load_weights(weight_file)
# 捕获 softmax 的输出，以便我们可以在可视化过程中解码输出
test_func = K.function([input_data], [y_pred])

viz_cb = VizCallback(run_name, test_func, img_gen.next_val())

model.fit_generator(
    generator=img_gen.next_train(),
    steps_per_epoch=(words_per_epoch - val_words) // minibatch_size,
    epochs=stop_epoch,
    validation_data=img_gen.next_val(),
    validation_steps=val_words // minibatch_size,
    callbacks=[viz_cb, img_gen],
    initial_epoch=start_epoch)

if __name__ == '__main__':
    run_name = datetime.datetime.now().strftime('%Y:%m:%d:%H:%M:%S')
    train(run_name, 0, 20, 128)
    # 增加到更宽的图像并从第 20 个轮次开始。

```

```
# 学到的重量会重新加载  
train(run_name, 20, 25, 512)
```

# 在 IMDB 情感分类任务上训练双向 LSTM。

Output after 4 epochs on CPU: ~0.8146. Time per epoch on CPU (Core i7): ~150s.

在 CPU 上经过 4 个轮次后的输出: ~0.8146。CPU (Core i7) 上每个轮次的时间: ~150s。

```
from __future__ import print_function
import numpy as np

from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Dropout, Embedding, LSTM, Bidirectional
from keras.datasets import imdb

max_features = 20000
# 在此数量的单词之后剪切文本 (取最常见的 max_features 个单词)
maxlen = 100
batch_size = 32

print('Loading data...')
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')

print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)
y_train = np.array(y_train)
y_test = np.array(y_test)

model = Sequential()
model.add(Embedding(max_features, 128, input_length=maxlen))
model.add(Bidirectional(LSTM(64)))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

# 尝试使用不同的优化器和优化器配置
model.compile('adam', 'binary_crossentropy', metrics=['accuracy'])

print('Train...')
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=4,
          validation_data=[x_test, y_test])
```



# 本示例演示了将 Convolution1D 用于文本分类。

2个轮次后达到 0.89 的测试精度。在 Intel i5 2.4Ghz CPU 上每轮次 90秒。在 Tesla K40 GPU 上每轮次 10秒。

```
from __future__ import print_function

from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalMaxPooling1D
from keras.datasets import imdb

# 设置参数:
max_features = 5000
 maxlen = 400
batch_size = 32
embedding_dims = 50
filters = 250
kernel_size = 3
hidden_dims = 250
epochs = 2

print('Loading data...')
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')

print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)

print('Build model...')
model = Sequential()

# 我们从有效的嵌入层开始，该层将 vocab 索引映射到 embedding_dims 维度
model.add(Embedding(max_features,
                    embedding_dims,
                    input_length=maxlen))
model.add(Dropout(0.2))

# 我们添加了一个 Convolution1D，它将学习大小为 filter_length 的过滤器词组过滤器：
model.add(Conv1D(filters,
                 kernel_size,
                 padding='valid',
                 activation='relu',
                 strides=1))

# 我们使用最大池化：
model.add(GlobalMaxPooling1D())
```

```
# We add a vanilla hidden layer:  
model.add(Dense(hidden_dims))  
model.add(Dropout(0.2))  
model.add(Activation('relu'))  
  
# 我们投影到单个单位输出层上，并用 sigmoid 压扁它：  
model.add(Dense(1))  
model.add(Activation('sigmoid'))  
  
model.compile(loss='binary_crossentropy',  
              optimizer='adam',  
              metrics=[ 'accuracy' ])  
model.fit(x_train, y_train,  
          batch_size=batch_size,  
          epochs=epochs,  
          validation_data=(x_test, y_test))
```



# 在 IMDB 情绪分类任务上训练循环卷积网络。

2 个轮次后达到 0.8498 的测试精度。K520 GPU 上为 41 秒/轮次。

```
from __future__ import print_function

from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Embedding
from keras.layers import LSTM
from keras.layers import Conv1D, MaxPooling1D
from keras.datasets import imdb

# Embedding
max_features = 20000
 maxlen = 100
embedding_size = 128

# Convolution
kernel_size = 5
filters = 64
pool_size = 4

# LSTM
lstm_output_size = 70

# Training
batch_size = 30
epochs = 2

...

注意：
batch_size 是高度敏感的
由于数据集非常小，因此仅需要 2 个轮次。
...

print('Loading data...')
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')

print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)

print('Build model...')

model = Sequential()
model.add(Embedding(max_features, embedding_size, input_length=maxlen))
model.add(Dropout(0.25))
```

```
model.add(Conv1D(filters,
                 kernel_size,
                 padding='valid',
                 activation='relu',
                 strides=1))
model.add(MaxPooling1D(pool_size=pool_size))
model.add(LSTM(lstm_output_size))
model.add(Dense(1))
model.add(Activation('sigmoid'))

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

print('Train...')
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          validation_data=(x_test, y_test))
score, acc = model.evaluate(x_test, y_test, batch_size=batch_size)
print('Test score:', score)
print('Test accuracy:', acc)
```



# 本示例演示了使用 fasttext 进行文本分类

根据Joulin等人的论文：

[Bags of Tricks for Efficient Text Classification](#)

在具有 uni-gram 和 bi-gram 嵌入的 IMDB 数据集上的结果：

Embedding	Accuracy, 5 epochs	Speed (s/epoch)	Hardware
Uni-gram	0.8813	8	i7 CPU
Bi-gram	0.9056	2	GTx 980M GPU

```
from __future__ import print_function
import numpy as np

from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Embedding
from keras.layers import GlobalAveragePooling1D
from keras.datasets import imdb

def create_ngram_set(input_list, ngram_value=2):
    """
    从整数列表中提取一组 n 元语法。
    >>> create_ngram_set([1, 4, 9, 4, 1, 4], ngram_value=2)
    {(4, 9), (4, 1), (1, 4), (9, 4)}
    >>> create_ngram_set([1, 4, 9, 4, 1, 4], ngram_value=3)
    [(1, 4, 9), (4, 9, 4), (9, 4, 1), (4, 1, 4)]
    """
    return set(zip(*[input_list[i:] for i in range(ngram_value)]))

def add_ngram(sequences, token indice, ngram_range=2):
    """
    通过附加 n-gram 值来增强列表（序列）的输入列表。
    示例：添加 bi-gram
    >>> sequences = [[1, 3, 4, 5], [1, 3, 7, 9, 2]]
    >>> token indice = {(1, 3): 1337, (9, 2): 42, (4, 5): 2017}
    >>> add_ngram(sequences, token indice, ngram_range=2)
    [[1, 3, 4, 5, 1337, 2017], [1, 3, 7, 9, 2, 1337, 42]]
```

```

示例：添加 tri-gram
>>> sequences = [[1, 3, 4, 5], [1, 3, 7, 9, 2]]
>>> token_indice = {(1, 3): 1337, (9, 2): 42, (4, 5): 2017, (7, 9, 2):
2018}
>>> add_ngram(sequences, token_indice, ngram_range=3)
[[[1, 3, 4, 5, 1337, 2017], [1, 3, 7, 9, 2, 1337, 42, 2018]]]
"""

new_sequences = []
for input_list in sequences:
    new_list = input_list[:]
    for ngram_value in range(2, ngram_range + 1):
        for i in range(len(new_list) - ngram_value + 1):
            ngram = tuple(new_list[i:i + ngram_value])
            if ngram in token_indice:
                new_list.append(token_indice[ngram])
    new_sequences.append(new_list)

return new_sequences

# 设置参数
# ngram_range = 2 会添加bi-grams 特征
ngram_range = 1
max_features = 20000
 maxlen = 400
batch_size = 32
embedding_dims = 50
epochs = 5

print('Loading data...')
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')
print('Average train sequence length: {}'.format(
    np.mean(list(map(len, x_train))), dtype=int)))
print('Average test sequence length: {}'.format(
    np.mean(list(map(len, x_test))), dtype=int)))

if ngram_range > 1:
    print('Adding {}-gram features'.format(ngram_range))
    # 从训练集中创建一组唯一的 n-gram。
    ngram_set = set()
    for input_list in x_train:
        for i in range(2, ngram_range + 1):
            set_of_ngram = create_ngram_set(input_list, ngram_value=i)
            ngram_set.update(set_of_ngram)

    # 将 n-gram token 映射到唯一整数的字典。
    # 整数值大于 max_features,
    # 以避免与现有功能冲突。
    start_index = max_features + 1
    token_indice = {v: k + start_index for k, v in enumerate(ngram_set)}
    indice_token = {token_indice[k]: k for k in token_indice}

    # max_features 是可以在数据集中找到的最大整数。
    max_features = np.max(list(indice_token.keys())) + 1

    # 使用 n-grams 功能增强 x_train 和 x_test

```

```
x_train = add_ngram(x_train, token indice, ngram_range)
x_test = add_ngram(x_test, token indice, ngram_range)
print('Average train sequence length: {}'.format(
    np.mean(list(map(len, x_train))), dtype=int)))
print('Average test sequence length: {}'.format(
    np.mean(list(map(len, x_test))), dtype=int)))

print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)

print('Build model...')
model = Sequential()

# 我们从有效的嵌入层开始，该层将 vocab 索引映射到 embedding_dims 维度
model.add(Embedding(max_features,
                    embedding_dims,
                    input_length=maxlen))

# 我们添加了 GlobalAveragePooling1D，它将对文档中所有单词执行平均嵌入
model.add(GlobalAveragePooling1D())

# 我们投影到单个单位输出层上，并用 sigmoid 压扁它：
model.add(Dense(1, activation='sigmoid'))

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

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          validation_data=(x_test, y_test))
```



# 在 IMDB 情感分类任务上训练 LSTM 模型。

与 TF-IDF + LogReg 之类的简单且快得多的方法相比，LSTM 实际上由于数据集太小而无济于事。

## 注意

- RNN 非常棘手。批次大小、损失和优化器的选择很重要，等等。某些配置无法收敛。
- 训练期间的 LSTM 损失减少模式可能与你在 CNN/MLP 等中看到的完全不同。

```
from __future__ import print_function

from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Embedding
from keras.layers import LSTM
from keras.datasets import imdb

max_features = 20000
# 在此数量的单词之后剪切文本（取最常见的 max_features 个单词）
 maxlen = 80
batch_size = 32

print('Loading data...')
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')

print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)

print('Build model...')
model = Sequential()
model.add(Embedding(max_features, 128))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))

# 尝试使用不同的优化器和优化器配置
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

print('Train...')
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=15,
          validation_data=(x_test, y_test))
score, acc = model.evaluate(x_test, y_test,
```

```
        batch_size=batch_size)
print('Test score:', score)
print('Test accuracy:', acc)
```



# Keras 序列到序列模型示例（字符级）。

该脚本演示了如何实现基本的字符级序列到序列模型。我们将其用于将英文短句逐个字符翻译成法语短句。请注意，进行字符级机器翻译是非常不寻常的，因为在此领域中词级模型更为常见。

## 算法总结

- 我们从一个领域的输入序列（例如英语句子）和另一个领域的对应目标序列（例如法语句子）开始；
- 编码器 LSTM 将输入序列变换为 2 个状态向量（我们保留最后的 LSTM 状态并丢弃输出）；
- 对解码器 LSTM 进行训练，以将目标序列转换为相同序列，但以后将偏移一个时间步，在这种情况下，该训练过程称为“教师强制”。它使用编码器的输出。实际上，解码器会根据输入序列，根据给定的 `targets[...t]` 来学习生成 `target[t+1...]`。
- 在推理模式下，当我们想解码未知的输入序列时，我们：
  - 对输入序列进行编码；
  - 从大小为1的目标序列开始（仅是序列开始字符）；
  - 将输入序列和 1 个字符的目标序列馈送到解码器，以生成下一个字符的预测；
  - 使用这些预测来采样下一个字符（我们仅使用 `argmax`）；
  - 将采样的字符附加到目标序列；
  - 重复直到我们达到字符数限制。

## 数据下载

[English to French sentence pairs.](#)

[Lots of neat sentence pairs datasets.](#)

## 参考

- [Sequence to Sequence Learning with Neural Networks](#)
- [Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation](#)

```
from __future__ import print_function

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

```

batch_size = 64 # 训练批次大小。
epochs = 100 # 训练迭代轮次。
latent_dim = 256 # 编码空间隐层维度。
num_samples = 10000 # 训练样本数。
# 磁盘数据文件路径。
data_path = 'fra-eng/fra.txt'

# 向量化数据。
input_texts = []
target_texts = []
input_characters = set()
target_characters = set()
with open(data_path, 'r', encoding='utf-8') as f:
    lines = f.read().split('\n')
for line in lines[: min(num_samples, len(lines) - 1)]:
    input_text, target_text = line.split('\t')
    # 我们使用 "tab" 作为 "起始序列" 字符,
    # 对于目标, 使用 "\n" 作为 "终止序列" 字符。
    target_text = '\t' + target_text + '\n'
    input_texts.append(input_text)
    target_texts.append(target_text)
    for char in input_text:
        if char not in input_characters:
            input_characters.add(char)
    for char in target_text:
        if char not in target_characters:
            target_characters.add(char)

input_characters = sorted(list(input_characters))
target_characters = sorted(list(target_characters))
num_encoder_tokens = len(input_characters)
num_decoder_tokens = len(target_characters)
max_encoder_seq_length = max([len(txt) for txt in input_texts])
max_decoder_seq_length = max([len(txt) for txt in target_texts])

print('Number of samples:', len(input_texts))
print('Number of unique input tokens:', num_encoder_tokens)
print('Number of unique output tokens:', num_decoder_tokens)
print('Max sequence length for inputs:', max_encoder_seq_length)
print('Max sequence length for outputs:', max_decoder_seq_length)

input_token_index = dict(
    [(char, i) for i, char in enumerate(input_characters)])
target_token_index = dict(
    [(char, i) for i, char in enumerate(target_characters)])

encoder_input_data = np.zeros(
    (len(input_texts), max_encoder_seq_length, num_encoder_tokens),
    dtype='float32')
decoder_input_data = np.zeros(
    (len(input_texts), max_decoder_seq_length, num_decoder_tokens),
    dtype='float32')
decoder_target_data = np.zeros(
    (len(input_texts), max_decoder_seq_length, num_decoder_tokens),
    dtype='float32')

```

```

for i, (input_text, target_text) in enumerate(zip(input_texts, target_texts)):
    for t, char in enumerate(input_text):
        encoder_input_data[i, t, input_token_index[char]] = 1.
    encoder_input_data[i, t + 1:, input_token_index[' ']] = 1.
    for t, char in enumerate(target_text):
        # decoder_target_data 领先 decoder_input_data by 一个时间步。
        decoder_input_data[i, t, target_token_index[char]] = 1.
        if t > 0:
            # decoder_target_data 将提前一个时间步，并且将不包含开始字符。
            decoder_target_data[i, t - 1, target_token_index[char]] = 1.
    decoder_input_data[i, t + 1:, target_token_index[' ']] = 1.
    decoder_target_data[i, t:, target_token_index[' ']] = 1.

# 定义输入序列并处理它。
encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
# 我们抛弃 `encoder_outputs`，只保留状态。
encoder_states = [state_h, state_c]

# 使用 `encoder_states` 作为初始状态来设置解码器。
decoder_inputs = Input(shape=(None, num_decoder_tokens))
# 我们将解码器设置为返回完整的输出序列，并返回内部状态。
# 我们不在训练模型中使用返回状态，但将在推理中使用它们。
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
                                      initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

# 定义模型，将 `encoder_input_data` & `decoder_input_data` 转换为
# `decoder_target_data`。
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

# 执行训练
model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
               metrics=['accuracy'])
model.fit([encoder_input_data, decoder_input_data], decoder_target_data,
          batch_size=batch_size,
          epochs=epochs,
          validation_split=0.2)
# 保存模型
model.save('s2s.h5')

# 接下来：推理模式（采样）。
# 这是演习：
# 1) 编码输入并检索初始解码器状态
# 2) 以该初始状态和 "序列开始" token 为目标运行解码器的一步。 输出将是下一个目标 token。
# 3) 重复当前目标 token 和当前状态

# 定义采样模型
encoder_model = Model(encoder_inputs, encoder_states)

decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
decoder_outputs, state_h, state_c = decoder_lstm(
    decoder_inputs, initial_state=decoder_states_inputs)

```

```

decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)
decoder_model = Model(
    [decoder_inputs] + decoder_states_inputs,
    [decoder_outputs] + decoder_states)

# 反向查询 token 索引可将序列解码回可读的内容。
reverse_input_char_index = dict(
    (i, char) for char, i in input_token_index.items())
reverse_target_char_index = dict(
    (i, char) for char, i in target_token_index.items())

def decode_sequence(input_seq):
    # 将输入编码为状态向量。
    states_value = encoder_model.predict(input_seq)

    # 生成长度为 1 的空目标序列。
    target_seq = np.zeros((1, 1, num_decoder_tokens))
    # 用起始字符填充目标序列的第一个字符。
    target_seq[0, 0, target_token_index['\t']] = 1.

    # 一批序列的采样循环
    # (为了简化，这里我们假设一批大小为 1)。
    stop_condition = False
    decoded_sentence = ''
    while not stop_condition:
        output_tokens, h, c = decoder_model.predict(
            [target_seq] + states_value)

        # 采样一个 token
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_char = reverse_target_char_index[sampled_token_index]
        decoded_sentence += sampled_char

        # 退出条件：达到最大长度或找到停止符。
        if (sampled_char == '\n' or
            len(decoded_sentence) > max_decoder_seq_length):
            stop_condition = True

        # 更新目标序列（长度为 1）。
        target_seq = np.zeros((1, 1, num_decoder_tokens))
        target_seq[0, 0, sampled_token_index] = 1.

        # 更新状态
        states_value = [h, c]

    return decoded_sentence

for seq_index in range(100):
    # 抽取一个序列（训练集的一部分）进行解码。
    input_seq = encoder_input_data[seq_index: seq_index + 1]
    decoded_sentence = decode_sequence(input_seq)
    print('-')
    print('Input sentence:', input_texts[seq_index])
    print('Decoded sentence:', decoded_sentence)

```



# 还原字符级序列到序列模型以生成预测。

该脚本载入由 [lstm\\_seq2seq.py](#) 保存的 `s2s.h5` 模型，并从中生成序列。它假设其未作任何改变（例如，`latent_dim`、输入数据和模型结构均不变）。

有关模型结构细节以及如何训练，参见 [lstm\\_seq2seq.py](#)。

```
from __future__ import print_function

from keras.models import Model, load_model
from keras.layers import Input
import numpy as np

batch_size = 64 # 训练批次大小。
epochs = 100 # 训练轮次数。
latent_dim = 256 # 编码空间隐层维度。
num_samples = 10000 # 训练样本数。
# 磁盘中数据文件路径。
data_path = 'fra-eng/fra.txt'

# 向量化数据。使用与训练脚本相同的方法。
# 注意：数据必须相同，以使字符->整数映射保持一致。
# 我们省略对 target_texts 的编码，因为不需要它们。
input_texts = []
target_texts = []
input_characters = set()
target_characters = set()
with open(data_path, 'r', encoding='utf-8') as f:
    lines = f.read().split('\n')
for line in lines[: min(num_samples, len(lines) - 1)]:
    input_text, target_text = line.split('\t')
    # 我们使用 "tab" 作为目标的 "开始序列" 字符，并使用 "\n" 作为 "结束序列" 字符。
    target_text = '\t' + target_text + '\n'
    input_texts.append(input_text)
    target_texts.append(target_text)
    for char in input_text:
        if char not in input_characters:
            input_characters.add(char)
    for char in target_text:
        if char not in target_characters:
            target_characters.add(char)

input_characters = sorted(list(input_characters))
target_characters = sorted(list(target_characters))
num_encoder_tokens = len(input_characters)
num_decoder_tokens = len(target_characters)
max_encoder_seq_length = max([len(txt) for txt in input_texts])
max_decoder_seq_length = max([len(txt) for txt in target_texts])

print('Number of samples:', len(input_texts))
print('Number of unique input tokens:', num_encoder_tokens)
print('Number of unique output tokens:', num_decoder_tokens)
```

```

print('Max sequence length for inputs:', max_encoder_seq_length)
print('Max sequence length for outputs:', max_decoder_seq_length)

input_token_index = dict(
    [(char, i) for i, char in enumerate(input_characters)])
target_token_index = dict(
    [(char, i) for i, char in enumerate(target_characters)])

encoder_input_data = np.zeros(
    (len(input_texts), max_encoder_seq_length, num_encoder_tokens),
    dtype='float32')

for i, input_text in enumerate(input_texts):
    for t, char in enumerate(input_text):
        encoder_input_data[i, t, input_token_index[char]] = 1.

# 恢复模型并构造编码器和解码器。
model = load_model('s2s.h5')

encoder_inputs = model.input[0]    # input_1
encoder_outputs, state_h_enc, state_c_enc = model.layers[2].output    # lstm_1
encoder_states = [state_h_enc, state_c_enc]
encoder_model = Model(encoder_inputs, encoder_states)

decoder_inputs = model.input[1]    # input_2
decoder_state_input_h = Input(shape=(latent_dim,), name='input_3')
decoder_state_input_c = Input(shape=(latent_dim,), name='input_4')
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
decoder_lstm = model.layers[3]
decoder_outputs, state_h_dec, state_c_dec = decoder_lstm(
    decoder_inputs, initial_state=decoder_states_inputs)
decoder_states = [state_h_dec, state_c_dec]
decoder_dense = model.layers[4]
decoder_outputs = decoder_dense(decoder_outputs)
decoder_model = Model(
    [decoder_inputs] + decoder_states_inputs,
    [decoder_outputs] + decoder_states)

# 反向查询 token 索引可将序列解码回可读的内容。
reverse_input_char_index = dict(
    (i, char) for char, i in input_token_index.items())
reverse_target_char_index = dict(
    (i, char) for char, i in target_token_index.items())

# 解码输入序列。未来的工作应支持波束搜索。
def decode_sequence(input_seq):
    # 将输入编码为状态向量。
    states_value = encoder_model.predict(input_seq)

    # 生成长度为 1 的空目标序列。
    target_seq = np.zeros((1, 1, num_decoder_tokens))
    # 用起始字符填充目标序列的第一个字符。
    target_seq[0, 0, target_token_index['\t']] = 1.

    # 一批序列的采样循环
    # (为了简化，这里我们假设一批大小为 1)。

```

```
stop_condition = False
decoded_sentence = ''
while not stop_condition:
    output_tokens, h, c = decoder_model.predict(
        [target_seq] + states_value)

    # 采样一个 token
    sampled_token_index = np.argmax(output_tokens[0, -1, :])
    sampled_char = reverse_target_char_index[sampled_token_index]
    decoded_sentence += sampled_char

    # 退出条件：达到最大长度或找到停止符。
    if (sampled_char == '\n' or
        len(decoded_sentence) > max_decoder_seq_length):
        stop_condition = True

    # 更新目标序列（长度为 1）。
    target_seq = np.zeros((1, 1, num_decoder_tokens))
    target_seq[0, 0, sampled_token_index] = 1.

    # 更新状态
    states_value = [h, c]

return decoded_sentence

for seq_index in range(100):
    # 抽取一个序列（训练集的一部分）进行解码。
    input_seq = encoder_input_data[seq_index: seq_index + 1]
    decoded_sentence = decode_sequence(input_seq)
    print('-')
    print('Input sentence:', input_texts[seq_index])
    print('Decoded sentence:', decoded_sentence)
```



# 如何使用有状态 LSTM 模型，有状态与无状态 LSTM 性能比较

## 有关 Keras LSTM 模型的更多文档

在输入/输出对上训练模型，其中输入是生成的长度为 `input_len` 的均匀分布随机序列，输出是窗口长度为 `tsteps` 的输入的移动平均值。`input_len` 和 `tsteps` 都在“可编辑参数”部分中定义。

较大的 `tsteps` 值意味着 LSTM 需要更多的内存来确定输入输出关系。该内存长度由 `lahead` 变量控制（下面有更多详细信息）。

其余参数：

- `input_len`：生成的输入序列的长度
- `lahead`：LSTM 针对每个输出点训练的输入序列长度
- `batch_size`, `epochs`：与 `model.fit(...)` 函数中的参数相同

当 `lahead > 1` 时，模型输入将预处理为数据的“滚动窗口视图”，窗口长度为 `lahead`。这类似于 `sklearn` 的 `view_as_windows`，其中 `window_shape` 是一个数字。

当 `lahead < tsteps` 时，只有有状态的 LSTM 会收敛，因为它的有状态性使其能够看到超出 `lahead` 赋予其的 `n` 点平均值的能力。无状态 LSTM 不具备此功能，因此受到其 `lahead` 参数的限制，该参数不足以查看 `n` 点平均值。

当 `lahead >= tsteps` 时，有状态和无状态 LSTM 都会收敛。

```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense, LSTM

# -----
# 可编辑参数
# 阅读脚本头中的文档以获取更多详细信息
# -----


# 输入长度
input_len = 1000

# 用于从训练 LSTM 的输入/输出对中的输入生成输出的移动平均值的窗口长度
# 例如，如果 tsteps=2, input=[1, 2, 3, 4, 5],
#       那么 output=[1.5, 2.5, 3.5, 4.5]
```

```

tsteps = 2

# LSTM 针对每个输出点训练的输入序列长度
lahead = 1

# 传递给 "model.fit(...)" 的训练参数
batch_size = 1
epochs = 10

# -----
# 主程序
# -----

print('*' * 33)
if lahead >= tsteps:
    print("STATELESS LSTM WILL ALSO CONVERGE")
else:
    print("STATELESS LSTM WILL NOT CONVERGE")
print('*' * 33)

np.random.seed(1986)

print('Generating Data...')

def gen_uniform_amp(amp=1, xn=10000):
    """生成 -amp 和 +amp 之间且长度为 xn 的均匀随机数据

    # 参数
        amp: 统一数据的最大/最小范围
        xn: 系列长度
    """
    data_input = np.random.uniform(-1 * amp, +1 * amp, xn)
    data_input = pd.DataFrame(data_input)
    return data_input

# 由于输出是输入的移动平均值,
# 因此输出的前几个点将是 NaN,
# 并且在训练 LSTM 之前将其从生成的数据中删除。
# 同样, 当 lahead > 1时, "滚动窗口视图"
# 后面的预处理步骤也将导致某些点丢失。
# 出于美学原因, 为了在预处理后保持生成的数据长度等于 input_len, 请添加一些点以说明将丢失的
# 值。
to_drop = max(tsteps - 1, lahead - 1)
data_input = gen_uniform_amp(amp=0.1, xn=input_len + to_drop)

# 将目标设置为输入的 N 点平均值
expected_output = data_input.rolling(window=tsteps, center=False).mean()

# 当 lahead > 1时, 需要将输入转换为 "滚动窗口视图"
# https://docs.scipy.org/doc/numpy/reference/generated/numpy.repeat.html
if lahead > 1:
    data_input = np.repeat(data_input.values, repeats=lahead, axis=1)
    data_input = pd.DataFrame(data_input)
    for i, c in enumerate(data_input.columns):
        data_input[c] = data_input[c].shift(i)

```

```

# 丢弃 nan
expected_output = expected_output[to_drop:]
data_input = data_input[to_drop:]

print('Input shape:', data_input.shape)
print('Output shape:', expected_output.shape)
print('Input head: ')
print(data_input.head())
print('Output head: ')
print(expected_output.head())
print('Input tail: ')
print(data_input.tail())
print('Output tail: ')
print(expected_output.tail())

print('Plotting input and expected output')
plt.plot(data_input[0][:10], '.')
plt.plot(expected_output[0][:10], '-')
plt.legend(['Input', 'Expected output'])
plt.title('Input')
plt.show()

def create_model(stateful):
    model = Sequential()
    model.add(LSTM(20,
                  input_shape=(lahead, 1),
                  batch_size=batch_size,
                  stateful=stateful))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam')
    return model

print('Creating Stateful Model...')
model_stateful = create_model(stateful=True)

# 切分训练/测试数据
def split_data(x, y, ratio=0.8):
    to_train = int(input_len * ratio)
    # 进行调整以匹配 batch_size
    to_train -= to_train % batch_size

    x_train = x[:to_train]
    y_train = y[:to_train]
    x_test = x[to_train:]
    y_test = y[to_train:]

    # 进行调整以匹配 batch_size
    to_drop = x.shape[0] % batch_size
    if to_drop > 0:
        x_test = x_test[:-1 * to_drop]
        y_test = y_test[:-1 * to_drop]

    # 一些重塑
    reshape_3 = lambda x: x.values.reshape((x.shape[0], x.shape[1], 1))
    x_train = reshape_3(x_train)

```

```

x_test = reshape_3(x_test)

reshape_2 = lambda x: x.values.reshape((x.shape[0], 1))
y_train = reshape_2(y_train)
y_test = reshape_2(y_test)

return (x_train, y_train), (x_test, y_test)

(x_train, y_train), (x_test, y_test) = split_data(data_input, expected_output)
print('x_train.shape: ', x_train.shape)
print('y_train.shape: ', y_train.shape)
print('x_test.shape: ', x_test.shape)
print('y_test.shape: ', y_test.shape)

print('Training')
for i in range(epochs):
    print('Epoch', i + 1, '/', epochs)
    # 请注意，批次 i 中样品 i 的最后状态将用作下一批中样品 i 的初始状态。
    # 因此，我们同时以低于 data_input 中包含的原始序列的分辨率对 batch_size 系列进行训练。
    # 这些系列中的每一个都偏移一个步骤，并且可以使用 data_input[i::batch_size] 提取。
    model_stateful.fit(x_train,
                        y_train,
                        batch_size=batch_size,
                        epochs=1,
                        verbose=1,
                        validation_data=(x_test, y_test),
                        shuffle=False)
    model_stateful.reset_states()

print('Predicting')
predicted_stateful = model_stateful.predict(x_test, batch_size=batch_size)

print('Creating Stateless Model...')
model_stateless = create_model(stateful=False)

print('Training')
model_stateless.fit(x_train,
                     y_train,
                     batch_size=batch_size,
                     epochs=epochs,
                     verbose=1,
                     validation_data=(x_test, y_test),
                     shuffle=False)

print('Predicting')
predicted_stateless = model_stateless.predict(x_test, batch_size=batch_size)

# -------

print('Plotting Results')
plt.subplot(3, 1, 1)
plt.plot(y_test)
plt.title('Expected')
plt.subplot(3, 1, 2)
# 删除第一个 "tsteps-1"，因为不可能预测它们，因为不存在要使用的 "上一个" 时间步

```

```
plt.plot((y_test - predicted_stateful).flatten()[tsteps - 1:])
plt.title('Stateful: Expected - Predicted')
plt.subplot(3, 1, 3)
plt.plot((y_test - predicted_stateless).flatten())
plt.title('Stateless: Expected - Predicted')
plt.show()
```



# 从尼采作品生成文本的示例脚本。

生成的文本开始听起来连贯之前，至少需要 20 个轮次。

建议在 GPU 上运行此脚本，因为循环网络的计算量很大。

如果在新数据上尝试使用此脚本，请确保您的语料库至少包含约 10 万个字符。~1M 更好。

```
from __future__ import print_function
from keras.callbacks import LambdaCallback
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.optimizers import RMSprop
from keras.utils.data_utils import get_file
import numpy as np
import random
import sys
import io

path = get_file(
    'nietzsche.txt',
    origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')
with io.open(path, encoding='utf-8') as f:
    text = f.read().lower()
print('corpus length:', len(text))

chars = sorted(list(set(text)))
print('total chars:', len(chars))
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))

# 以 maxlen 字符的半冗余序列剪切文本
maxlen = 40
step = 3
sentences = []
next_chars = []
for i in range(0, len(text) - maxlen, step):
    sentences.append(text[i: i + maxlen])
    next_chars.append(text[i + maxlen])
print('nb sequences:', len(sentences))

print('Vectorization...')
x = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        x[i, t, char_indices[char]] = 1
        y[i, char_indices[next_chars[i]]] = 1

# 建立模型：单个 LSTM
```

```

print('Build model...')
model = Sequential()
model.add(LSTM(128, input_shape=( maxlen, len(chars) )))
model.add(Dense(len(chars), activation='softmax'))

optimizer = RMSprop(learning_rate=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)

def sample(preds, temperature=1.0):
    # 辅助函数从概率数组中采样索引
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)

def on_epoch_end(epoch, _):
    # 在每个轮次结束时调用的函数。 打印生成的文本。
    print()
    print('----- Generating text after Epoch: %d' % epoch)

    start_index = random.randint(0, len(text) - maxlen - 1)
    for diversity in [0.2, 0.5, 1.0, 1.2]:
        print('----- diversity:', diversity)

        generated = ''
        sentence = text[start_index: start_index + maxlen]
        generated += sentence
        print('----- Generating with seed: "' + sentence + '"')
        sys.stdout.write(generated)

        for i in range(400):
            x_pred = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(sentence):
                x_pred[0, t, char_indices[char]] = 1.

            preds = model.predict(x_pred, verbose=0)[0]
            next_index = sample(preds, diversity)
            next_char = indices_char[next_index]

            sentence = sentence[1:] + next_char

            sys.stdout.write(next_char)
            sys.stdout.flush()
        print()

print_callback = LambdaCallback(on_epoch_end=on_epoch_end)

model.fit(x, y,
           batch_size=128,
           epochs=60,
           callbacks=[print_callback])

```



# 在 MNIST 数据集上训练辅助分类器 GAN (ACGAN)。

[有关辅助分类器 GAN 的更多详细信息。](#)

你应该在大约 5 个轮次后开始看到合理的图像，而在大约 15 个轮次后开始看到良好的图像。你应该使用 GPU，因为大量卷积运算在 CPU 上非常慢。如果你打算进行迭代，请首选 TensorFlow 后端，因为使用 Theano 的话编译时间可能会成为阻碍。

耗时：

硬件	后端	Time / Epoch
CPU	TF	3 hrs
Titan X (maxwell)	TF	4 min
Titan X (maxwell)	TH	7 min

[有关更多信息和示例输出，请咨询 Keras 中的辅助分类器生成对抗网络。](#)

```
from __future__ import print_function

from collections import defaultdict
try:
    import cPickle as pickle
except ImportError:
    import pickle
from PIL import Image

from six.moves import range

from keras.datasets import mnist
from keras import layers
from keras.layers import Input, Dense, Reshape, Flatten, Embedding, Dropout
from keras.layers import BatchNormalization
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import Conv2DTranspose, Conv2D
from keras.models import Sequential, Model
from keras.optimizers import Adam
from keras.utils.generic_utils import Progbar
import numpy as np

np.random.seed(1337)
num_classes = 10
```

```

def build_generator(latent_size):
    # 我们将一对 (z, L) 映射到图像空间 (... , 28, 28, 1), 其中 z 是隐向量, L 是从 P_c
    # 绘制的标签。
    cnn = Sequential()

    cnn.add(Dense(3 * 3 * 384, input_dim=latent_size, activation='relu'))
    cnn.add(Reshape((3, 3, 384)))

    # 上采样至 (7, 7, ...)
    cnn.add(Conv2DTranspose(192, 5, strides=1, padding='valid',
                           activation='relu',
                           kernel_initializer='glorot_normal'))
    cnn.add(BatchNormalization())

    # 上采样至 (14, 14, ...)
    cnn.add(Conv2DTranspose(96, 5, strides=2, padding='same',
                           activation='relu',
                           kernel_initializer='glorot_normal'))
    cnn.add(BatchNormalization())

    # 上采样至 (28, 28, ...)
    cnn.add(Conv2DTranspose(1, 5, strides=2, padding='same',
                           activation='tanh',
                           kernel_initializer='glorot_normal'))

    # 这是 GAN 论文中通常提到的 z 空间
    latent = Input(shape=(latent_size,))

    # 这将是我们的标签
    image_class = Input(shape=(1,), dtype='int32')

    cls = Embedding(num_classes, latent_size,
                    embeddings_initializer='glorot_normal')(image_class)

    # z 空间和一类条件嵌入之间的 hadamard 积
    h = layers.multiply([latent, cls])

    fake_image = cnn(h)

    return Model([latent, image_class], fake_image)

def build_discriminator():
    # 根据参考文献中的建议, 使用 LeakyReLU 构建相对标准的转换网络
    cnn = Sequential()

    cnn.add(Conv2D(32, 3, padding='same', strides=2,
                  input_shape=(28, 28, 1)))
    cnn.add(LeakyReLU(0.2))
    cnn.add(Dropout(0.3))

    cnn.add(Conv2D(64, 3, padding='same', strides=1))
    cnn.add(LeakyReLU(0.2))
    cnn.add(Dropout(0.3))

```

```

cnn.add(Conv2D(128, 3, padding='same', strides=2))
cnn.add(LeakyReLU(0.2))
cnn.add(Dropout(0.3))

cnn.add(Conv2D(256, 3, padding='same', strides=1))
cnn.add(LeakyReLU(0.2))
cnn.add(Dropout(0.3))

cnn.add(Flatten())

image = Input(shape=(28, 28, 1))

features = cnn(image)

# 第一个输出 (name=generation) 是鉴别是否认为所显示的图像是伪造的,
# 而第二个输出 (name=auxiliary) 是鉴别认为图像所属的类。
fake = Dense(1, activation='sigmoid', name='generation')(features)
aux = Dense(num_classes, activation='softmax', name='auxiliary')(features)

return Model(image, [fake, aux])

if __name__ == '__main__':
    # 论文的批次和潜在大小
    epochs = 100
    batch_size = 100
    latent_size = 100

    # https://arxiv.org/abs/1511.06434 建议的 Adam 参数
    adam_lr = 0.0002
    adam_beta_1 = 0.5

    # 建立鉴别器
    print('Discriminator model:')
    discriminator = build_discriminator()
    discriminator.compile(
        optimizer=Adam(learning_rate=adam_lr, beta_1=adam_beta_1),
        loss=['binary_crossentropy', 'sparse_categorical_crossentropy']
    )
    discriminator.summary()

    # 建立生成器
    generator = build_generator(latent_size)

    latent = Input(shape=(latent_size, ))
    image_class = Input(shape=(1,), dtype='int32')

    # 取得假图片
    fake = generator([latent, image_class])

    # 我们只希望能够训练组合模型的生成
    discriminator.trainable = False
    fake, aux = discriminator(fake)
    combined = Model([latent, image_class], [fake, aux])

    print('Combined model:')
    combined.compile()

```

```

        optimizer=Adam(learning_rate=adam_lr, beta_1=adam_beta_1),
        loss=['binary_crossentropy', 'sparse_categorical_crossentropy']
    )
combined.summary()

# 获取我们的 mnist 数据，并强制其形状为 (... , 28, 28, 1)，范围为 [-1, 1]
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = (x_train.astype(np.float32) - 127.5) / 127.5
x_train = np.expand_dims(x_train, axis=-1)

x_test = (x_test.astype(np.float32) - 127.5) / 127.5
x_test = np.expand_dims(x_test, axis=-1)

num_train, num_test = x_train.shape[0], x_test.shape[0]

train_history = defaultdict(list)
test_history = defaultdict(list)

for epoch in range(1, epochs + 1):
    print('Epoch {}/{}'.format(epoch, epochs))

    num_batches = int(np.ceil(x_train.shape[0] / float(batch_size)))
    progress_bar = Progbar(target=num_batches)

    epoch_gen_loss = []
    epoch_disc_loss = []

    for index in range(num_batches):
        # 得到一批真实的图像
        image_batch = x_train[index * batch_size:(index + 1) * batch_size]
        label_batch = y_train[index * batch_size:(index + 1) * batch_size]

        # 产生一批新的噪音
        noise = np.random.uniform(-1, 1, (len(image_batch), latent_size))

        # 从 p_c 采样一些标签
        sampled_labels = np.random.randint(0, num_classes,
                                         len(image_batch))

        # 使用生成的标签作为调节器，生成一批假图像。
        # 我们将采样的标签重塑为 (len(image_batch), 1),
        # 以便我们可以将它们作为一个序列的长度送入嵌入层
        generated_images = generator.predict(
            [noise, sampled_labels.reshape((-1, 1))], verbose=0)

        x = np.concatenate((image_batch, generated_images))

        # 使用单面 soft real/fake 标签
        # Salimans et al., 2016
        # https://arxiv.org/pdf/1606.03498.pdf (Section 3.4)
        soft_zero, soft_one = 0, 0.95
        y = np.array(
            [soft_one] * len(image_batch) + [soft_zero] *
            len(image_batch))
        aux_y = np.concatenate((label_batch, sampled_labels), axis=0)

        # 我们不希望鉴别器也能最大化生成图像上辅助分类器的分类精度,

```

```

# 因此我们不训练鉴别器为生成图像生成类标签（请参阅 https://openreview.net/forum?id=rJXTf9Bxg）。
# 为了保留辅助分类器的样本权重总和，我们将样本权重 2 分配给实际图像。
disc_sample_weight = [np.ones(2 * len(image_batch)),
                      np.concatenate((np.ones(len(image_batch)) * 2,

```

$$\text{len}(\text{image\_batch})))]$$

```

# 看看鉴别器是否能弄清楚自己...
epoch_disc_loss.append(discriminator.train_on_batch(
    x, [y, aux_y], sample_weight=disc_sample_weight))

# 制造新的声音。我们在这里生成 2 倍批量大小，
# 这样我们就可以使生成器对与鉴别器相同数量的图像进行优化
noise = np.random.uniform(-1, 1, (2 * len(image_batch),
latent_size))
sampled_labels = np.random.randint(0, num_classes, 2 *
len(image_batch))

# 我们想训练生成器来欺骗鉴别器
# 对于生成器，我们希望所有 {fake, not-fake} 标签都说不假
trick = np.ones(2 * len(image_batch)) * soft_one

epoch_gen_loss.append(combined.train_on_batch(
    [noise, sampled_labels.reshape((-1, 1))],
    [trick, sampled_labels]))

progress_bar.update(index + 1)

print('Testing for epoch {}:{}.'.format(epoch))

# 在这里评估测试损失

# 产生一批新的噪音
noise = np.random.uniform(-1, 1, (num_test, latent_size))

# 从 p_c 采样一些标签并从中生成图像
sampled_labels = np.random.randint(0, num_classes, num_test)
generated_images = generator.predict(
    [noise, sampled_labels.reshape((-1, 1))], verbose=False)

x = np.concatenate((x_test, generated_images))
y = np.array([1] * num_test + [0] * num_test)
aux_y = np.concatenate((y_test, sampled_labels), axis=0)

# 看看鉴别器是否能弄清楚自己...
discriminator_test_loss = discriminator.evaluate(
    x, [y, aux_y], verbose=False)

discriminator_train_loss = np.mean(np.array(epoch_disc_loss), axis=0)

# 制造新的噪声
noise = np.random.uniform(-1, 1, (2 * num_test, latent_size))
sampled_labels = np.random.randint(0, num_classes, 2 * num_test)

trick = np.ones(2 * num_test)

```

```

generator_test_loss = combined.evaluate(
    [noise, sampled_labels.reshape((-1, 1))],
    [trick, sampled_labels], verbose=False)

generator_train_loss = np.mean(np.array(epoch_gen_loss), axis=0)

# 生成有关性能的轮次报告
train_history['generator'].append(generator_train_loss)
train_history['discriminator'].append(discriminator_train_loss)

test_history['generator'].append(generator_test_loss)
test_history['discriminator'].append(discriminator_test_loss)

print('{0:<22s} | {1:4s} | {2:15s} | {3:5s}'.format(
    'component', *discriminator.metrics_names))
print('-' * 65)

ROW_FMT = '{0:<22s} | {1:<4.2f} | {2:<15.4f} | {3:<5.4f}'
print(ROW_FMT.format('generator (train)',
                     *train_history['generator'][-1]))
print(ROW_FMT.format('generator (test)',
                     *test_history['generator'][-1]))
print(ROW_FMT.format('discriminator (train)',
                     *train_history['discriminator'][-1]))
print(ROW_FMT.format('discriminator (test)',
                     *test_history['discriminator'][-1]))

# 在每个轮次保存权重
generator.save_weights(
    'params_generator_epoch_{0:03d}.hdf5'.format(epoch), True)
discriminator.save_weights(
    'params_discriminator_epoch_{0:03d}.hdf5'.format(epoch), True)

# 生成一些数字来显示
num_rows = 40
noise = np.tile(np.random.uniform(-1, 1, (num_rows, latent_size)),
               (num_classes, 1))

sampled_labels = np.array([
    [i] * num_rows for i in range(num_classes)
]).reshape(-1, 1)

# 批量显示
generated_images = generator.predict(
    [noise, sampled_labels], verbose=0)

# 准备按类别标签排序的真实图像
real_labels = y_train[(epoch - 1) * num_rows * num_classes:
                      epoch * num_rows * num_classes]
indices = np.argsort(real_labels, axis=0)
real_images = x_train[(epoch - 1) * num_rows * num_classes:
                      epoch * num_rows * num_classes][indices]

# 显示生成的图像，白色分隔符，真实图像
img = np.concatenate(
    (generated_images,

```

```
np.repeat(np.ones_like(x_train[:1]), num_rows, axis=0),
real_images))

# 将它们排列成网格
img = (np.concatenate([r.reshape(-1, 28)
                      for r in np.split(img, 2 * num_classes + 1)
                      ], axis=-1) * 127.5 + 127.5).astype(np.uint8)

Image.fromarray(img).save(
    'plot_epoch_{0:03d}_generated.png'.format(epoch))

with open('acgan-history.pkl', 'wb') as f:
    pickle.dump({'train': train_history, 'test': test_history}, f)
```