




Keras: 基于 Python 的深度学习库
Keras: The Python Deep Learning library*

Author: Keras-Team

Contributor: 万震 (WAN Zhen)

 wanzhenchn

 wanzhen@cqu.edu.cn

2018 年 12 月 24 日

前言

整理 [Keras: 基于 Python 的深度学习库](#) PDF 版的主要原因在于学习 [Keras](#) 深度学习库时方便本地查阅，下载最新 PDF 版本请访问: <https://github.com/wanzhenchn/keras-docs-zh>。

感谢 [keras-team](#) 所做的中文翻译工作，本文档制作基于此处。

严正声明：本文档可免费用于学习和科学研究，可自由传播，但切勿擅自用于商业用途，由此引发一切后果贡献者概不负责。

The main reason of organizing PDF version based the [Chinese Keras Markdown](#) is that it is easy to **read locally** when learning the Keras Deep Learning Library. For the latest PDF version, please visit <https://github.com/wanzhenchn/keras-docs-zh>.

Thanks for the Chinese translation work done by [keras-team](#), this document is produced based on it.

Statement: This document can be freely used for learning and scientific research and is freely disseminated, but it must not be used for commercial purposes. Otherwise, the contributor is not responsible for the consequences.

目录

1 Keras: 基于 Python 的深度学习库	1
1.1 你恰好发现了 Keras	1
1.2 指导原则	1
1.3 快速开始: 30 秒上手 Keras	1
1.4 安装指引	3
1.5 使用 TensorFlow 以外的后端	3
1.6 技术支持	3
1.7 为什么取名为 Keras?	4
2 为什么选择 Keras?	5
2.1 Keras 优先考虑开发人员的经验	5
2.2 Keras 被工业界和学术界广泛采用	5
2.3 Keras 可以轻松将模型转化为产品	6
2.4 Keras 支持多个后端引擎, 并且不会将你锁定到一个生态系统中	6
2.5 Keras 拥有强大的多 GPU 和分布式训练支持	6
2.6 Keras 的发展得到深度学习生态系统中的关键公司的支持	7
3 快速开始	8
3.1 Sequential 顺序模型指引	8
3.1.1 开始使用 Keras 顺序 (Sequential) 模型	8
3.1.2 指定输入数据的尺寸	8
3.1.3 编译	9
3.1.4 训练	9
3.1.5 例子	10
3.1.5.1 基于多层感知器 (MLP) 的 softmax 多分类:	11
3.1.5.2 基于多层感知器的二分类:	12
3.1.5.3 类似 VGG 的卷积神经网络:	12
3.1.5.4 基于 LSTM 的序列分类:	13
3.1.5.5 基于 1D 卷积的序列分类:	14
3.1.5.6 基于栈式 LSTM 的序列分类	14
3.1.5.7 带有状态 (stateful) 的相同的栈式 LSTM 模型	15
3.2 函数式 API 指引	17
3.2.1 开始使用 Keras 函数式 API	17
3.2.2 例一: 全连接网络	17
3.2.3 所有的模型都可调用, 就像网络层一样	17
3.2.4 多输入多输出模型	18
3.2.5 共享网络层	20
3.2.6 层「节点」的概念	21

3.2.7	更多的例子	22
3.2.7.1	Inception 模型	22
3.2.7.2	卷积层上的残差连接	23
3.2.7.3	共享视觉模型	23
3.2.7.4	视觉问答模型	24
3.2.7.5	视频问答模型	25
3.3	Keras FAQ: 常见问题解答	26
3.3.1	Keras FAQ: 常见问题解答	26
3.3.2	如何引用 Keras?	26
3.3.3	如何在 GPU 上运行 Keras?	26
3.3.4	如何在多 GPU 上运行 Keras 模型?	27
3.3.4.1	数据并行	27
3.3.4.2	设备并行	27
3.3.5	“sample”, “batch”, “epoch” 分别是什么?	28
3.3.6	如何保存 Keras 模型?	28
3.3.6.1	保存/加载整个模型 (结构 + 权重 + 优化器状态)	28
3.3.6.2	只保存/加载模型的结构	29
3.3.6.3	只保存/加载模型的权重	29
3.3.6.4	处理已保存模型中的自定义层 (或其他自定义对象)	30
3.3.7	为什么训练误差比测试误差高很多?	31
3.3.8	如何获取中间层的输出?	31
3.3.9	如何用 Keras 处理超过内存的数据集?	32
3.3.10	在验证集的误差不再下降时, 如何中断训练?	32
3.3.11	验证集划分是如何计算的?	32
3.3.12	在训练过程中数据是否会混洗?	32
3.3.13	如何在每个 epoch 后记录训练集和验证集的误差和准确率?	32
3.3.14	如何「冻结」网络层?	33
3.3.15	如何使用有状态 RNN (stateful RNNs)?	33
3.3.16	如何从 Sequential 模型中移除一个层?	34
3.3.17	如何在 Keras 中使用预训练的模型?	35
3.3.18	如何在 Keras 中使用 HDF5 输入?	35
3.3.19	Keras 配置文件保存在哪里?	36
3.3.20	如何在 Keras 开发过程中获取可复现的结果?	36
3.3.21	如何在 Keras 中安装 HDF5 或 h5py 来保存我的模型?	37
4	模型	39
4.1	关于 Keras 模型	39
4.2	Sequential 顺序模型 API	41
4.2.1	Sequential 顺序模型 API	41
4.2.2	常用 Sequential 属性	41

4.2.3	Sequential 模型方法	41
4.2.3.1	compile	41
4.2.3.2	fit	42
4.2.3.3	evaluate	43
4.2.3.4	predict	44
4.2.3.5	train_on_batch	44
4.2.3.6	test_on_batch	45
4.2.3.7	predict_on_batch	45
4.2.3.8	fit_generator	45
4.2.3.9	evaluate_generator	47
4.2.3.10	predict_generator	47
4.2.3.11	get_layer	48
4.3	函数式 API	49
4.3.1	Model 类 API	49
4.3.2	Model 的实用属性	49
4.3.3	Model 类模型方法	49
4.3.3.1	compile	49
4.3.3.2	fit	50
4.3.3.3	evaluate	51
4.3.3.4	predict	52
4.3.3.5	train_on_batch	52
4.3.3.6	test_on_batch	53
4.3.3.7	predict_on_batch	53
4.3.3.8	fit_generator	54
4.3.3.9	evaluate_generator	55
4.3.3.10	predict_generator	56
4.3.3.11	get_layer	57
5	关于 Keras 网络层	58
5.1	关于 Keras 层	58
5.2	核心网络层	59
5.2.1	Dense [source]	59
5.2.2	Activation [source]	60
5.2.3	Dropout [source]	60
5.2.4	Flatten [source]	60
5.2.5	Input [source]	61
5.2.6	Reshape [source]	62
5.2.7	Permute [source]	62
5.2.8	RepeatVector [source]	63
5.2.9	Lambda [source]	63

5.2.10	ActivityRegularization [source]	64
5.2.11	Masking [source]	65
5.3	卷积层 Convolutional	66
5.3.1	Conv1D [source]	66
5.3.2	Conv2D [source]	67
5.3.3	SeparableConv2D [source]	68
5.3.4	Conv2DTranspose [source]	70
5.3.5	Conv3D [source]	71
5.3.6	Cropping1D [source]	72
5.3.7	Cropping2D [source]	73
5.3.8	Cropping3D [source]	74
5.3.9	UpSampling1D [source]	75
5.3.10	UpSampling2D [source]	75
5.3.11	UpSampling3D [source]	76
5.3.12	ZeroPadding1D [source]	76
5.3.13	ZeroPadding2D [source]	77
5.3.14	ZeroPadding3D [source]	78
5.4	池化层 Pooling	79
5.4.1	MaxPooling1D [source]	79
5.4.2	MaxPooling2D [source]	79
5.4.3	MaxPooling3D [source]	80
5.4.4	AveragePooling1D [source]	80
5.4.5	AveragePooling2D [source]	81
5.4.6	AveragePooling3D [source]	82
5.4.7	GlobalMaxPooling1D [source]	82
5.4.8	GlobalAveragePooling1D [source]	83
5.4.9	GlobalMaxPooling2D [source]	83
5.4.10	GlobalAveragePooling2D [source]	83
5.4.11	GlobalMaxPooling3D [source]	84
5.4.12	GlobalAveragePooling3D [source]	84
5.5	局部连接层 Locally-connected	86
5.5.1	LocallyConnected1D [source]	86
5.5.2	LocallyConnected2D [source]	87
5.6	循环层 Recurrent	89
5.6.1	RNN [source]	89
5.6.2	SimpleRNN [source]	91
5.6.3	GRU [source]	92
5.6.4	LSTM [source]	94
5.6.5	ConvLSTM2D [source]	95

5.6.6	SimpleRNNCell [source]	97
5.6.7	GRUCell [source]	98
5.6.8	LSTMCell [source]	99
5.6.9	StackedRNNCells [source]	100
5.6.10	CuDNNNGRU [source]	100
5.6.11	CuDNNLSTM [source]	101
5.7	嵌入层 Embedding	103
5.7.1	Embedding [source]	103
5.8	融合层 Merge	104
5.8.1	Add [source]	104
5.8.2	Subtract [source]	104
5.8.3	Multiply [source]	105
5.8.4	Average [source]	105
5.8.5	Maximum [source]	105
5.8.6	Concatenate [source]	105
5.8.7	Dot [source]	105
5.8.8	add	106
5.8.9	subtract	106
5.8.10	multiply	107
5.8.11	average	107
5.8.12	maximum	107
5.8.13	concatenate	108
5.8.14	dot	108
5.9	高级激活层 Advanced Activations	109
5.9.1	LeakyReLU [source]	109
5.9.2	PReLU [source]	109
5.9.3	ELU [source]	110
5.9.4	ThresholdedReLU [source]	110
5.9.5	Softmax [source]	110
5.9.6	ReLU[source]	111
5.10	标准化层 Normalization	112
5.10.1	BatchNormalization [source]	112
5.11	噪声层 Noise	113
5.11.1	GaussianNoise [source]	113
5.11.2	GaussianDropout [source]	113
5.11.3	AlphaDropout [source]	114
5.12	层封装器 wrappers	115
5.12.1	TimeDistributed [source]	115
5.12.2	Bidirectional [source]	115

5.13 编写你自己的 Keras 层	117
6 数据预处理	118
6.1 序列预处理	118
6.1.1 TimeseriesGenerator	118
6.1.2 pad_sequences	119
6.1.3 skipgrams	120
6.1.4 make_sampling_table	121
6.2 文本预处理	122
6.2.1 Tokenizer	122
6.2.2 hashing_trick	122
6.2.3 one_hot	123
6.2.4 text_to_word_sequence	123
6.3 图像预处理	125
6.3.1 ImageDataGenerator 类	125
6.3.2 ImageDataGenerator 类方法	129
6.3.2.1 apply_transform	129
6.3.2.2 fit	129
6.3.2.3 flow	130
6.3.2.4 flow_from_directory	131
6.3.2.5 get_random_transform	132
6.3.2.6 random_transform	132
6.3.2.7 standardize	133
7 损失函数 Losses	134
7.1 损失函数的使用	134
7.2 可用损失函数	134
7.2.1 mean_squared_error	134
7.2.2 mean_absolute_error	134
7.2.3 mean_absolute_percentage_error	134
7.2.4 mean_squared_logarithmic_error	134
7.2.5 squared_hinge	134
7.2.6 hinge	134
7.2.7 categorical_hinge	135
7.2.8 logcosh	135
7.2.9 categorical_crossentropy	135
7.2.10 sparse_categorical_crossentropy	135
7.2.11 binary_crossentropy	135
7.2.12 kullback_leibler_divergence	135
7.2.13 poisson	135

7.2.14	cosine_proximity	135
8	评估标准 Metrics	137
8.1	评价函数的用法	137
8.1.1	参数	137
8.1.2	返回值	137
8.2	可使用的评价函数	137
8.2.1	binary_accuracy	137
8.2.2	categorical_accuracy	137
8.2.3	sparse_categorical_accuracy	137
8.2.4	top_k_categorical_accuracy	137
8.2.5	sparse_top_k_categorical_accuracy	138
8.3	自定义评价函数	138
9	优化器 Optimizers	139
9.1	优化器的用法	139
9.2	Keras 优化器的公共参数	139
9.2.1	SGD [source]	139
9.2.2	RMSprop [source]	140
9.2.3	Adagrad [source]	140
9.2.4	Adadelata [source]	141
9.2.5	Adam [source]	141
9.2.6	Adamax [source]	141
9.2.7	Nadam [source]	142
9.2.8	TFOptimizer [source]	142
10	激活函数 Activations	143
10.1	激活函数的用法	143
10.2	预定义激活函数	143
10.2.1	softmax	143
10.2.2	elu	143
10.2.3	selu	144
10.2.4	softplus	144
10.2.5	softsign	144
10.2.6	relu	144
10.2.7	tanh	144
10.2.8	sigmoid	144
10.2.9	hard_sigmoid	144
10.2.10	linear	144
10.3	高级激活函数	145

11 回调函数 Callbacks	146
11.1 回调函数使用	146
11.1.1 Callback [source]	146
11.1.2 BaseLogger [source]	146
11.1.3 TerminateOnNaN [source]	146
11.1.4 ProgbarLogger [source]	146
11.1.5 History [source]	147
11.1.6 ModelCheckpoint [source]	147
11.1.7 EarlyStopping [source]	148
11.1.8 RemoteMonitor [source]	148
11.1.9 LearningRateScheduler [source]	148
11.1.10 TensorBoard [source]	149
11.1.11 ReduceLROnPlateau [source]	149
11.1.12 CSVLogger [source]	150
11.1.13 LambdaCallback [source]	150
11.2 创建一个回调函数	152
11.2.1 例: 记录损失历史	152
11.2.2 例: 模型检查点	152
12 常用数据集 Datasets	154
12.1 CIFAR10 小图像分类数据集	154
12.2 CIFAR100 小图像分类数据集	154
12.3 IMDB 电影评论情感分类数据集	154
12.4 路透社新闻主题分类	155
12.5 MNIST 手写字符数据集	156
12.6 Fashion-MNIST 时尚物品数据集	156
12.7 Boston 房价回归数据集	157
13 预训练模型 Applications	158
13.1 可用的模型	158
13.2 图像分类模型的示例代码	158
13.2.1 使用 ResNet50 进行 ImageNet 分类	158
13.2.2 使用 VGG16 提取特征	159
13.2.3 从 VGG19 的任意中间层中抽取特征	159
13.2.4 在新类上微调 InceptionV3	160
13.2.5 通过自定义输入 tensor 构建 InceptionV3	161
13.3 模型概览	162
13.3.1 Xception	162
13.3.2 VGG16	163
13.3.3 VGG19	164

13.3.4 ResNet50	165
13.3.5 InceptionV3	165
13.3.6 InceptionResNetV2	166
13.3.7 MobileNet	167
13.3.8 DenseNet	168
13.3.9 NASNet	169
14 后端 Backend	171
14.1 什么是「后端」?	171
14.2 从一个后端切换到另一个后端	171
14.3 keras.json 详细配置	171
14.4 使用抽象 Keras 后端编写新代码	172
14.5 后端函数	173
15 初始化 Initializers	226
15.1 初始化器的用法	226
15.2 可用的初始化器	226
15.2.1 Initializer [source]	226
15.2.2 Zeros [source]	226
15.2.3 Ones [source]	226
15.2.4 Constant [source]	226
15.2.5 RandomNormal [source]	227
15.2.6 RandomUniform [source]	227
15.2.7 TruncatedNormal [source]	227
15.2.8 VarianceScaling [source]	227
15.2.9 Orthogonal [source]	228
15.2.10 Identity [source]	228
15.2.11 lecun_uniform	228
15.2.12 glorot_normal	229
15.2.13 glorot_uniform	229
15.2.14 he_normal	230
15.2.15 lecun_normal	230
15.2.16 he_uniform	230
15.3 使用自定义初始化器	231
16 正则化 Regularizers	232
16.1 正规化的使用	232
16.2 例子	232
16.3 可用的惩罚	232
16.4 开发新的正则化器	232

17 约束 Constraints	233
17.1 约束项的使用	233
17.2 可用的约束	233
18 可视化 Visualization	234
19 Scikit-learn API	235
20 工具	236
20.1 CustomObjectScope [source]	236
20.2 HDF5Matrix [source]	236
20.3 Sequence [source]	236
20.4 to_categorical	237
20.5 normalize	238
20.6 get_file	238
20.7 print_summary	239
20.8 plot_model	239
20.9 multi_gpu_model	239
21 贡献	242
21.1 关于 Github Issues 和 Pull Requests	242
21.2 漏洞报告	242
21.3 请求新功能	242
21.4 请求贡献代码	243
21.5 Pull Requests 合并请求	243
21.6 添加新的样例	244

1 Keras: 基于 Python 的深度学习库

1.1 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](#)。

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

1.2 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**。新的模块是很容易添加的（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。

1.3 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，它是由多个网络层线性堆叠的栈。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 顺序模型如下所示：

```
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, 或者其他的任何模型, 就是这么的快。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** 顺序模型
- 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。

1.4 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 (用于可视化工具绘制模型图)。

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):**

```
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
```

1.5 使用 TensorFlow 以外的后端

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。

1.6 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。

你也可以在 [Github issues](#) 中张贴漏洞报告和新功能请求 (仅限于此)。注意请先阅读[规范文档](#)。

1.7 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).

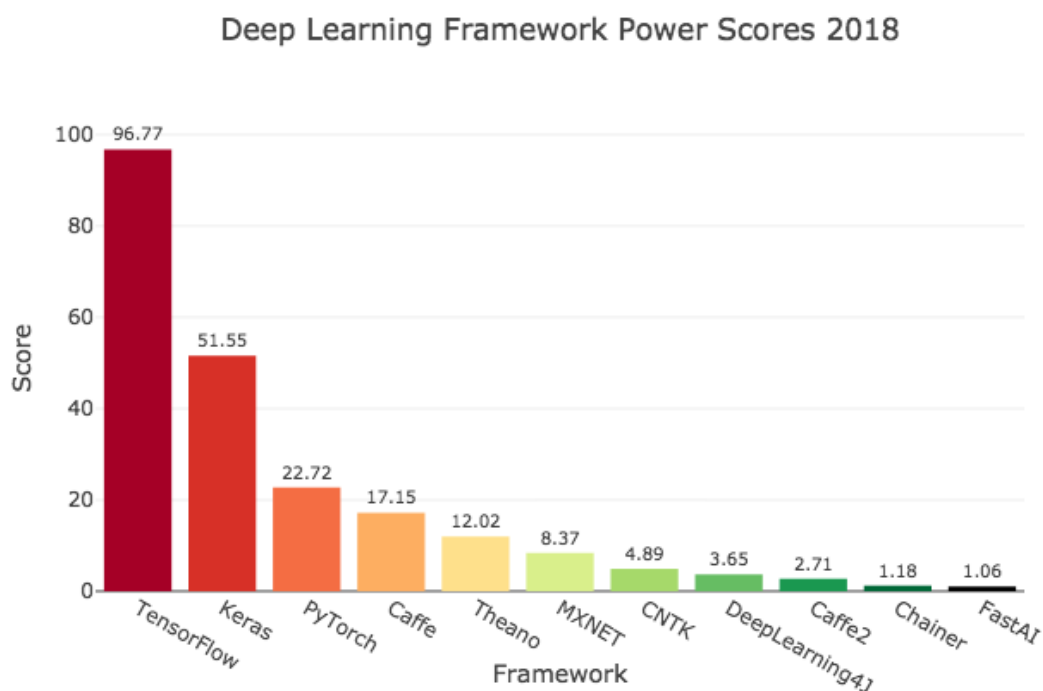
2 为什么选择 Keras?

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。

2.1 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。

2.2 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)。

您已经不断与使用 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，在上载到预印本服务器 [arXiv.org](https://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。

2.3 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](#)。
- 在安卓上，通过 TensorFlow Android runtime，例如：[Not Hotdog app](#)。
- 在浏览器上，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 网页应用后端](#)（比如 [Flask app](#)）中。
- 在 JVM，通过 [SkyMind](#) 提供的 [DL4J](#) 模型导入。
- 在 Raspberry Pi 树莓派上。

2.4 Keras 支持多个后端引擎，并且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：用一种后端训练模型，再将它载入另一种后端中（比如为了发布）。支持的后端有：

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

亚马逊也正在为 Keras 开发 MXNet 后端。

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 GPU](#)。
- [Google TPU](#)，通过 TensorFlow 后端和 Google Cloud。
- OpenGL 支持的 GPU, 比如 AMD, 通过 [PlaidML Keras 后端](#)。

2.5 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 模型有第一流的支持。
- Keras 模型可以被转换为 TensorFlow 估计器并在 [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。

2.6 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。

3 快速开始

3.1 Sequential 顺序模型指引

3.1.1 开始使用 Keras 顺序 (Sequential) 模型

顺序模型是多个网络层的线性堆叠。

你可以通过将层的列表传递给 `Sequential` 的构造函数，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'))
```

3.1.2 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（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（一个整数或 `None` 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))
```

3.1.3 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。

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])
```

3.1.4 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)
```

3.1.5 例子

这里有几个可以帮助你开始的例子！

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 和 CNN
- 基于 LSTM 的字符级文本生成

... 等等。

3.1.5.1 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)
```

3.1.5.2 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)
```

3.1.5.3 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)
```

3.1.5.4 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

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)
```

3.1.5.5 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  
  
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)
```

3.1.5.6 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）。

```
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))

```

3.1.5.7 带有状态 (stateful) 的相同的栈式 LSTM 模型

有状态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))
```

3.2 函数式 API 指引

3.2.1 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例子开始。

3.2.2 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
```

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

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

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

```
# 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
```

3.2.3 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)
```

3.2.4 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的时间等。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：

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

# 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：

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

3.2.5 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。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 (140, 256) 的矩阵，即每条推特 140 字符，每个字符为 256 维的 one-hot 编码（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=(140, 256))
tweet_b = Input(shape=(140, 256))

# 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。

3.2.6 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？

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

```
a = Input(shape=(140, 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=(140, 256))
b = Input(shape=(140, 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，并且将由 `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')
convded_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)

convded_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)
```

3.2.7 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。

3.2.7.1 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)
```

3.2.7.2 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])
```

3.2.7.3 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=(28, 28, 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=(28, 28, 1))
digit_b = Input(shape=(28, 28, 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)
```

3.2.7.4 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])
```

然后在上面对一个 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。

3.2.7.5 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)
```

3.3 Keras FAQ: 常见问题解答

3.3.1 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?
- 为什么训练误差比测试误差高很多?
- 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洗?
- 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?
- 如何使用有状态 RNN (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?

3.3.2 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}},  
}
```

3.3.3 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'
```

3.3.4 如何在多 GPU 上运行 Keras 模型?

我们建议使用 TensorFlow 后端。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。

3.3.4.1 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。这里是一个简单的例子：

```
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)
```

3.3.4.2 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)

```

3.3.5 “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，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 越大就越近似。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 evaluation_data 或 evaluation_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）。

3.3.6 如何保存 Keras 模型？

3.3.6.1 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）。

例子：

```
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')
```

3.3.6.2 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)
```

3.3.6.3 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)
```

例如：

```
"""
```

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)
```

3.3.6.4 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')
```



```
# 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]
```

3.3.9 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。

3.3.10 在验证集的误差不再下降时，如何中断训练？

你可以使用 `EarlyStopping` 回调函数：

```
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](#)。

3.3.11 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）。

3.3.12 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。

3.3.13 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)
```

3.3.14 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
```

3.3.15 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

例子：

```
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。

3.3.16 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"
```

3.3.17 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
- Inception v3
- Inception-ResNet v2
- MobileNet v1

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

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

有关一些简单的用法示例，请参阅 [应用模块的文档](#)。

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](#)

3.3.18 如何在 Keras 中使用 HDF5 输入？

你可以使用 `keras.utils.io_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)
```

3.3.19 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。默认的配置文件的如下所示：

```
{
  "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（`channel_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。

3.3.20 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。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 Python 3.2.3 以上所必需的，
# 为了使某些基于散列的操作可复现。
# https://docs.python.org/3.4/using/cmdline.html#envvar-PYTHONHASHSEED
# https://github.com/keras-team/keras/issues/2280#issuecomment-306959926
```



```

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

# 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的来源。
# 更多详情，见: https://stackoverflow.com/questions/42022950/which-seeds-have-to-be-set-when

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 ...

```

3.3.21 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。

4 模型

4.1 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)
# or, for Sequential:
model = Sequential.from_config(config)
```

- `model.get_weights()`: 返回模型权重的张量列表，类型为 Numpy array。
- `model.set_weights(weights)`: 从 Numpy array 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，只包含结构。你可以通过以下代码，从 JSON 字符串中重新实例化相同的模型（带有重新初始化的权重）：

```
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（带有重新初始化的权重）：

```
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 子类

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` 子类 API 引入于 Keras 2.2.0)。

这里是一个用 `Model` 子类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)。

在子类模型中, 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子类模型:

- `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 子类化 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。

4.2 Sequential 顺序模型 API

4.2.1 Sequential 顺序模型 API

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

4.2.2 常用 Sequential 属性

- `model.layers` 是添加到模型的层的列表。

4.2.3 Sequential 模型方法

4.2.3.1 compile

```
compile(self, optimizer, loss, metrics=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。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'}`。
- **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。

例

```
model = Sequential()
model.add(Dense(32, input_shape=(500,)))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

4.2.3.2 fit

```
fit(self, x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None,
    validation_split=0.0, validation_data=None, shuffle=True, class_weight=None,
    sample_weight=None, initial_epoch=0, steps_per_epoch=None, validation_steps=None)
```

以固定数量的轮次（数据集上的迭代）训练模型。

参数

- **x**: 训练数据的 Numpy 数组。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。如果从本地框架张量馈送（例如 TensorFlow 数据张量）数据, `x` 可以是 `None`（默认）。
- **y**: 目标（标签）数据的 Numpy 数组。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。如果从本地框架张量馈送（例如 TensorFlow 数据张量）数据, `y` 可以是 `None`（默认）。
- **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。
- **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**: 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_split**: 在 0 和 1 之间浮动。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。
- **validation_data**: 元组 (`x_val`, `y_val`) 或元组 (`x_val`, `y_val`, `val_sample_weights`), 用来评估损失, 以及在每轮结束时的任何模型度量指标。模型将不会在这个数据上进行训练。这个参数会覆盖 `validation_split`。

- **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）。您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"**。
- **initial_epoch**: 开始训练的轮次（有助于恢复之前的训练）。
- **steps_per_epoch**: 在声明一个轮次完成并开始下一个轮次之前的总步数（样品批次）。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**: 只有在指定了 **steps_per_epoch** 时才有用。停止前要验证的总步数（批次样本）。

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。

4.2.3.3 evaluate

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

计算一些输入数据的误差，逐批次。

参数

- **x**: 输入数据，Numpy 数组或列表（如果模型有多输入）。如果从本地框架张量馈送（例如 TensorFlow 数据张量）数据，x 可以是 **None**（默认）。
- **y**: 标签，Numpy 数组。如果从本地框架张量馈送（例如 TensorFlow 数据张量）数据，y 可以是 **None**（默认）。
- **batch_size**: 整数。每次梯度更新的样本数。如果未指定，默认为 32。
- **verbose**: 日志显示模式，0 或 1。
- **sample_weight**: 样本权重，Numpy 数组。
- **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**。

返回

标量测试误差（如果模型没有评估指标）或标量列表（如果模型计算其他指标）。属性 `model.metrics_names` 将提供标量输出的显示标签。

异常

- **RuntimeError**: 如果模型从未编译。

4.2.3.4 predict

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

为输入样本生成输出预测。

输入样本逐批处理。

参数

- **x**: 输入数据, Numpy 数组。
- **batch_size**: 整数。如未指定, 默认为 32。
- **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）。默认值 `None`。

返回

预测的 Numpy 数组。

4.2.3.5 train_on_batch

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

一批样品的单次梯度更新。

Arguments

- **x**: 输入数据, Numpy 数组或列表（如果模型有多输入）。
- **y**: 标签, Numpy 数组。
- **class_weight**: 将类别映射为权重的字典, 用于在训练时缩放损失函数。
- **sample_weight**: 样本权重, Numpy 数组。

返回

标量训练误差（如果模型没有评估指标）或标量列表（如果模型计算其他指标）。属性 `model.metrics_names` 将提供标量输出的显示标签。

异常

- **RuntimeError**: 如果模型从未编译。

4.2.3.6 test_on_batch

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

在一批样本上评估模型。

参数

- **x**: 输入数据, Numpy 数组或列表 (如果模型有多输入)。
- **y**: 标签, Numpy 数组。
- **sample_weight**: 样本权重, Numpy 数组。

返回

标量测试误差 (如果模型没有评估指标) 或标量列表 (如果模型计算其他指标)。属性 `model.metrics_names` 将提供标量输出的显示标签。

异常

- **RuntimeError**: 如果模型从未编译。

4.2.3.7 predict_on_batch

```
predict_on_batch(self, 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。

4.2.3.8 fit_generator

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

使用 Python 生成器逐批生成的数据, 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。

参数

- **generator**: 一个生成器。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。所有的数组都必须包含同样数量的样本。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）。它通常应该等于你的数据集的样本数量除以批量大小。可选参数 **Sequence**: 如果未指定，将使用 `len(generator)` 作为步数。
- **epochs**: 整数，数据的迭代总轮数。请注意，与 `initial_epoch` 一起，参数 `epochs` 应被理解为「最终轮数」。模型并不是训练了 `epochs` 轮，而是到第 `epochs` 轮停止训练。
- **verbose**: 日志显示模式。0, 1 或 2。
- **callbacks**: 在训练时调用的一系列回调函数。
- **validation_data**: 它可以是以下之一：
 - 验证数据的生成器
 - 一个 (inputs, targets) 元组
 - 一个 (inputs, targets, sample_weights) 元组。
- **validation_steps**: 仅当 `validation_data` 是一个生成器时才可用。每个 epoch 结束时验证集生成器产生的步数。它通常应该等于你的数据集的样本数量除以批量大小。可选参数 **Sequence**: 如果未指定，将使用 `len(generator)` 作为步数。
- **class_weight**: 将类别映射为权重的字典。
- **max_queue_size**: 生成器队列的最大尺寸。
- **workers**: 使用的最大进程数量。
- **use_multiprocessing**: 如果 `True`，则使用基于进程的多线程。请注意，因为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。
- **shuffle**: 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。
- **initial_epoch**: 开始训练的轮次（有助于恢复之前的训练）。

返回

一个 History 对象。

异常

- **RuntimeError**: 如果模型从未编译。

例

```
def generate_arrays_from_file(path):
    while 1:
        f = open(path)
        for line in f:
            # create Numpy arrays of input data
            # and labels, from each line in the file
            x, y = process_line(line)
            yield (x, y)
```

```
f.close()
```

```
model.fit_generator(generate_arrays_from_file('/my_file.txt'),  
                    steps_per_epoch=1000, epochs=10)
```

4.2.3.9 evaluate_generator

```
evaluate_generator(self, generator, steps=None, max_queue_size=10, workers=1,  
                  use_multiprocessing=False)
```

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**: 生成器，生成 (inputs, targets) 或 (inputs, targets, sample_weights)
- **steps**: 在停止之前，来自 **generator** 的总步数 (样本批次)。可选参数 **Sequence**: 如果未指定，将使用 `len(generator)` 作为步数。
- **max_queue_size**: 生成器队列的最大尺寸。
- **workers**: 使用的最大进程数量。
- **use_multiprocessing**: 如果 `True`，则使用基于进程的多线程。请注意，因为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。

返回

标量测试误差（如果模型没有评估指标）或标量列表（如果模型计算其他指标）。属性 `model.metrics_names` 将提供标量输出的显示标签。

异常

- **RuntimeError**: 如果模型从未编译。

4.2.3.10 predict_generator

```
predict_generator(self, generator, steps=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**: 返回批量输入样本的生成器。
- **steps**: 在停止之前，来自 **generator** 的总步数 (样本批次)。可选参数 **Sequence**: 如果未指定，将使用 `len(generator)` 作为步数。
- **max_queue_size**: 生成器队列的最大尺寸。
- **workers**: 使用的最大进程数量。

- **use_multiprocessing**: 如果 True, 则使用基于进程的多线程。请注意, 因为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。

4.2.3.11 get_layer

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

提取模型的某一层。

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。

4.3 函数式 API

4.3.1 Model 类 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](#)。

4.3.2 Model 的实用属性

- `model.layers` 是包含模型图的层的展平的列表。
- `model.inputs` 是输入张量的列表。
- `model.outputs` 是输出张量的列表。

4.3.3 Model 类模型方法

4.3.3.1 compile

```
compile(self, optimizer, loss, 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。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'}`。

- **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。如果是张量，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。

4.3.3.2 fit

```
fit(self, x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None,
    validation_split=0.0, validation_data=None, shuffle=True,
    class_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None,
    validation_steps=None)
```

以固定数量的轮次（数据集上的迭代）训练模型。

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。如果从本地框架张量馈送（例如 TensorFlow 数据张量）数据，**x** 可以是 **None**（默认）。
- **y**: 目标（标签）数据的 Numpy 数组（如果模型只有一个输出），或者是 Numpy 数组的列表（如果模型有多个输出）。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。如果从本地框架张量馈送（例如 TensorFlow 数据张量）数据，**y** 可以是 **None**（默认）。
- **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。
- **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:** 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_split:** 在 0 和 1 之间浮动。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。
- **validation_data:** 元组 (`x_val`, `y_val`) 或元组 (`x_val`, `y_val`, `val_sample_weights`)，用来评估损失，以及在每轮结束时的任何模型度量指标。模型将不会在这个数据上进行训练。这个参数会覆盖 `validation_split`。
- **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）。您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"`。
- **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（样品批次）。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:** 只有在指定了 `steps_per_epoch` 时才有用。停止前要验证的总步数（批次样本）。

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。

4.3.3.3 evaluate

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

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**: 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。如果从本地框架张量馈送（例如 TensorFlow 数据张量）数据，x 可以是 None（默认）。
- **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。如果从本地框架张量馈送（例如 TensorFlow 数据张量）数据，y 可以是 None（默认）。
- **batch_size**: 整数或 None。每次评估的样本数。如果未指定，默认为 32。
- **verbose**: 0 或 1。日志显示模式。0 = 安静模式，1 = 进度条。
- **sample_weight**: 测试样本的可选 Numpy 权重数组，用于对损失函数进行加权。您可以传递与输入样本长度相同的扁平（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，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。

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。

4.3.3.4 predict

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

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，Numpy 数组（或者 Numpy 数组的列表，如果模型有多个输出）。
- **batch_size**: 整数。如未指定，默认为 32。
- **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）。默认值 None。

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。

4.3.3.5 train_on_batch

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


运行一批样品的单次梯度更新。

`__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。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。

Returns

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。

4.3.3.6 test_on_batch

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

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。

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。

4.3.3.7 predict_on_batch

```
predict_on_batch(self, 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)。

4.3.3.8 fit_generator

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

使用 Python 生成器逐批生成的数据, 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。
- **epochs**: 整数, 数据的迭代总轮数。
- **verbose**: 日志显示模式。0, 1 或 2。
- **callbacks**: 在训练时调用的一系列回调函数。
- **validation_data**: 它可以是以下之一:
 - 验证数据的生成器
 - 一个 `(inputs, targets)` 元组
 - 一个 `(inputs, targets, sample_weights)` 元组。
- **validation_steps**: 仅当 `validation_data` 是一个生成器时才可用。在停止前 `generator` 生成的总步数 (样本批数)。对于 `Sequence`, 它是可选的: 如果未指定, 将使用 `len(generator)` 作为步数。

- **class_weight**: 将类别索引映射为权重的字典。
- **max_queue_size**: 整数。生成器队列的最大尺寸。如未指定, `max_queue_size` 将默认为 10。
- **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。如未指定, `workers` 将默认为 False。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。
- **shuffle**: 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。
- **initial_epoch**: 开始训练的轮次 (有助于恢复之前的训练)。

返回

一个 `History` 对象。

例

```
def generate_arrays_from_file(path):
    while 1:
        f = open(path)
        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)
```

异常

- **ValueError**: 如果生成器生成的数据格式不正确。

4.3.3.9 evaluate_generator

```
evaluate_generator(self, generator, steps=None, max_queue_size=10, workers=1,
                  use_multiprocessing=False)
```

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。
- **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。

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。

4.3.3.10 predict_generator

```
predict_generator(self, generator, steps=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。
- **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。

4.3.3.11 get_layer

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

根据名称（唯一）或索引值查找网络层。

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。

5 关于 Keras 网络层

5.1 关于 Keras 层

所有 Keras 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)`

5.2 核心网络层

5.2.1 Dense [source]

```
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)
```

就是普通的全连接层。

Dense 实现以下操作: $\text{output} = \text{activation}(\text{dot}(\text{input}, \text{kernel}) + \text{bias})$ 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。

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 (see [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: $(\text{batch_size}, \dots, \text{input_dim})$ 。最常见的情况是一个尺寸为 $(\text{batch_size}, \text{input_dim})$ 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)。

5.2.2 Activation [\[source\]](#)

```
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: [hyperref\[activations\]](#) 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。

5.2.3 Dropout [\[source\]](#)

```
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](#)

5.2.4 Flatten [\[source\]](#)

```
keras.layers.Flatten()
```

将输入展平。不影响批量大小。

例


```

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

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

```

5.2.5 Input [\[source\]](#)

`keras.engine.topology.Input()`

`Input()` 用于实例化 Keras 张量。

Keras 张量是底层后端 (Theano, TensorFlow or 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。A shape tuple (integer), not including the batch size. 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（不可以重用同一个名字两次）。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。

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)

```

5.2.6 Reshape [\[source\]](#)

```
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`

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)
```

5.2.7 Permute [\[source\]](#)

```
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。

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。

5.2.8 RepeatVector [\[source\]](#)

`keras.layers.RepeatVector(n)`

将输入重复 `n` 次。

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)。

5.2.9 Lambda [\[source\]](#)

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

将任意表达式封装为 Layer 对象。

例

```
# 添加一个  $x \rightarrow x^2$  层
model.add(Lambda(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))
```

参数

- **function:** 需要封装的函数。将输入张量作为第一个参数。
- **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)`
- **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)。

5.2.10 ActivityRegularization [\[source\]](#)

```
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。

5.2.11 Masking [\[source\]](#)

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

使用覆盖值覆盖序列，以跳过时间步。

对于输入张量的每一个时间步（张量的第一个维度），如果所有时间步中输入张量的值与 `mask_value` 相等，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。

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 和 #5，因为你缺乏这几个时间步的数据。你可以：

- 设置 `x[:, 3, :] = 0.` 以及 `x[:, 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))
```

5.3 卷积层 Convolutional

5.3.1 Conv1D [\[source\]](#)

```
keras.layers.Conv1D(filters, kernel_size, strides=1,
                    padding='valid', dilation_rate=1, activation=None,
                    use_bias=True, kernel_initializer='glorot_uniform',
                    bias_initializer='zeros', kernel_regularizer=None,
                    bias_regularizer=None, activity_regularizer=None,
                    kernel_constraint=None, bias_constraint=None)
```

1D 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`），例如，`(10, 128)` 表示 10 个 128 维的向量组成的向量序列，`(None, 128)` 表示 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!= 1 与指定 `dilation_rate` 值!= 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。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](#)。
- **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!= 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

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。

5.3.2 Conv2D [\[source\]](#)

```
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，不包含样本表示的轴），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。

5.3.3 SeparableConv2D [\[source\]](#)

```
keras.layers.SeparableConv2D(filters, kernel_size, strides=(1, 1), padding='valid',
                              data_format=None, depth_multiplier=1, activation=None,
                              use_bias=True, depthwise_initializer='glorot_uniform',
                              pointwise_initializer='glorot_uniform', bias_initializer='zeros',
                              depthwise_regularizer=None, pointwise_regularizer=None,
                              bias_regularizer=None, activity_regularizer=None,
                              depthwise_constraint=None, pointwise_constraint=None, bias_constraint=None)
```

深度方向的可分离 2D 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作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”。
- **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 [hyperref\[constraints\]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。

5.3.4 Conv2DTranspose [source]

```
keras.layers.Conv2DTranspose(filters, kernel_size, strides=(1, 1), padding='valid',
                             data_format=None, activation=None, use_bias=True,
                             kernel_initializer='glorot_uniform', bias_initializer='zeros',
                             kernel_regularizer=None, bias_regularizer=None,
                             activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
```

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, 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 `(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。

参考文献

- [A guide to convolution arithmetic for deep learning](#)
- [Deconvolutional Networks](#)

5.3.5 Conv3D [\[source\]](#)

```
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，不包含样本表示的轴），例如，`input_shape=(128, 128, 128, 1)` 表示 128x128x128 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。

5.3.6 Cropping1D [\[source\]](#)

```
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)。

5.3.7 Cropping2D [\[source\]](#)

```
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_first'，输出 4D 张量，尺寸为 (batch, filters, new_rows, new_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, 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, 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)。

例子

```
# 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)
```

5.3.8 Cropping3D [\[source\]](#)

```
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、高度和宽度的 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

- 如果 `data_format` 为 "channels_last", 输入 5D 张量, 尺寸为 (batch, first_axis_to_crop, second_axis_to_crop, third_axis_to_crop, depth)。
- 如果 `data_format` 为 "channels_first", 输入 5D 张量, 尺寸为 (batch, depth, first_axis_to_crop, second_axis_to_crop, third_axis_to_crop)。

输出尺寸

- 如果 `data_format` 为 `"channels_last"`, 输出 5D 张量, 尺寸为 `(batch, first_cropped_axis, second_cropped_axis, third_cropped_axis, depth)`
- 如果 `data_format` 为 `"channels_first"`, 输出 5D 张量, 尺寸为 `(batch, depth, first_cropped_axis, second_cropped_axis, third_cropped_axis)`。

5.3.9 UpSampling1D [\[source\]](#)

```
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)`。

5.3.10 UpSampling2D [\[source\]](#)

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

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"`。

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)。

5.3.11 UpSampling3D [\[source\]](#)

```
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)。

5.3.12 ZeroPadding1D [\[source\]](#)

```
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。

- 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)。

5.3.13 ZeroPadding2D [\[source\]](#)

```
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：解释为高度和高度的 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)。

5.3.14 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

- 如果 data_format 为 "channels_last", 输入 5D 张量, 尺寸为 (batch, first_axis_to_pad, second_axis_to_pad, third_axis_to_pad, depth)。
- 如果 data_format 为 "channels_first", 输入 5D 张量, 尺寸为 (batch, depth, first_axis_to_pad, second_axis_to_pad, third_axis_to_pad)。

输出尺寸

- 如果 data_format 为 "channels_last", 输出 5D 张量, 尺寸为 (batch, first_padded_axis, second_padded_axis, third_axis_to_pad, depth)。
- 如果 data_format 为 "channels_first", 输出 5D 张量, 尺寸为 (batch, depth, first_padded_axis, second_padded_axis, third_axis_to_pad)。

5.4 池化层 Pooling

5.4.1 MaxPooling1D [\[source\]](#)

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

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）。

输入尺寸

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

输出尺寸

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

5.4.2 MaxPooling2D [\[source\]](#)

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

对于空域数据的最大池化。

参数

- **pool_size**: 整数，或者 2 个整数元组，（垂直方向，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，整数元组或者是 None。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**: 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

- 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

5.4.3 MaxPooling3D [\[source\]](#)

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

对于 3D（空域，或时空域）数据的最大池化。

参数

- **pool_size**: 3 个整数的元组，缩小（维度 1，维度 2，维度 3）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 个整数的元组，或者是 None。步长值。
- **padding**: "valid" 或者 "same"（区分大小写）。
- **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, 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

5.4.4 AveragePooling1D [\[source\]](#)

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

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）。

输入尺寸

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

输出尺寸

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

5.4.5 AveragePooling2D [\[source\]](#)

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

对于空域数据的平均池化。

参数

- **pool_size**: 整数，或者 2 个整数元组，（垂直方向，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，整数元组或者是 None。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**: 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

- 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

5.4.6 AveragePooling3D [\[source\]](#)

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

对于 3D（空域，或者时空域）数据的平均池化。

参数

- **pool_size**: 3 个整数的元组，缩小（维度 1，维度 2，维度 3）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 个整数的元组，或者是 None。步长值。
- **padding**: "valid" 或者 "same"（区分大小写）。
- **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, 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

5.4.7 GlobalMaxPooling1D [\[source\]](#)

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

对于时序数据的全局最大池化。

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。

5.4.8 GlobalAveragePooling1D [\[source\]](#)

`keras.layers.GlobalAveragePooling1D()`

对于时序数据的全局平均池化。

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。

5.4.9 GlobalMaxPooling2D [\[source\]](#)

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

对于空域数据的全局最大池化。

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

5.4.10 GlobalAveragePooling2D [\[source\]](#)

`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

5.4.11 GlobalMaxPooling3D [\[source\]](#)

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

对于 3D 数据的全局最大池化。

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, 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

5.4.12 GlobalAveragePooling3D [\[source\]](#)

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

对于 3D 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, 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

5.5 局部连接层 Locally-connected

5.5.1 LocallyConnected1D [\[source\]](#)

```
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，在输入的不同部分应用不同的一组过滤器。

例子

```
# 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 值!= 1 与指定 dilation_rate 值!= 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。
- **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。

5.5.2 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，在输入的不同部分应用不同的一组过滤器。

例子

```
# 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。

5.6 循环层 Recurrent

5.6.1 RNN [\[source\]](#)

```
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项目的类:
 - 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用可选参数 `constants`，详见下面的小节“关于传递外部常量的注意事项”。
- 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）。这也可以是整数的列表/元组（每个状态一个大小）。在这种情况下，第一项（`state_size [0]`）应该与单元输出的大小相同。`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** 为 True，则返回张量列表。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)`。
- 否则，返回尺寸为 `(batch_size, units)` 的 2D 张量。

屏蔽覆盖

该层支持以可变数量的时间步长对输入数据进行屏蔽覆盖。要将屏蔽引入数据，请使用 `Embedding` 层，并将 `mask_zero` 参数设置为 `True`。

关于在 RNN 中使用状态的注意事项

你可以将 RNN 层设置为 `stateful`（有状态的），这意味着针对一批中的样本计算的状态将被重新用作下一批样品的初始状态。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 初始状态的注意事项

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 传递外部常量的注意事项

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。

例子

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)

```

5.6.2 SimpleRNN [\[source\]](#)

```

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)

```

完全连接的 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](#))。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，但它往往会占用更多的内存。展开只适用于短序列。

5.6.3 GRU [\[source\]](#)

```
keras.layers.GRU(units, activation='tanh', recurrent_activation='hard_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=1,
                  return_sequences=False, return_state=False,
                  go_backwards=False, stateful=False, unroll=False, reset_after=False)
```

门限循环单元网络 - Cho et al. 2014.

有两种变体。默认的基于 1406.1078v3 并且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 并且顺序倒置。

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`)。
- **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，但它往往会占用更多的内存。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](#)

5.6.4 LSTM [\[source\]](#)

```
keras.layers.LSTM(units, 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,
                  dropout=0.0, recurrent_dropout=0.0,
                  implementation=1, return_sequences=False,
                  return_state=False, go_backwards=False, stateful=False,
                  unroll=False)
```

长短期记忆网络层 - 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](#))。
- **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.](#)
- **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](#) (original 1997 paper)
- [Learning to forget: Continual prediction with LSTM](#)
- [Supervised sequence labeling with recurrent neural networks](#)
- [A Theoretically Grounded Application of Dropout in Recurrent Neural Networks](#)

5.6.5 ConvLSTM2D [\[source\]](#)

```
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!= 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, 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 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 $\neq 1$ 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.](#)。
- **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'`，返回 5D 张量，尺寸为: (samples, time, channels, rows, cols)。
- 如果 `data_format='channels_last'`，返回 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 依赖于过滤器的尺寸和填充。

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。

5.6.6 SimpleRNNCell [\[source\]](#)

```
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](#))。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。

5.6.7 GRUCell [\[source\]](#)

```
keras.layers.GRUCell(units, activation='tanh', recurrent_activation='hard_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=1)
```

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](#))。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](#))。
- **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。

5.6.8 LSTMCell [\[source\]](#)

```
keras.layers.LSTMCell(units, 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, implementation=1)
```

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](#))。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.](#)
- **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。

5.6.9 StackedRNNCells [\[source\]](#)

```
keras.layers.StackedRNNCells(cells)
```

允许将一堆 RNN 单元表现为一个单元的封装器。

用于实现高效堆叠的 RNN。

参数

- **cells:** RNN 单元实例的列表。

例子

```
cells = [  
    keras.layers.LSTMCell(output_dim),  
    keras.layers.LSTMCell(output_dim),  
    keras.layers.LSTMCell(output_dim),  
]  
  
inputs = keras.Input((timesteps, input_dim))  
x = keras.layers.RNN(cells)(inputs)
```

5.6.10 CuDNNGRU [\[source\]](#)

```
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。

5.6.11 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](#))。
- **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.](#)
- **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](#))。
- **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。

5.7 嵌入层 Embedding

5.7.1 Embedding [\[source\]](#)

```
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。

例子

```
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](#))。
- **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](#)

5.8 融合层 Merge

5.8.1 Add [\[source\]](#)

`keras.layers.Add()`

计算一个列表的输入张量的和。

相加层接受一个列表的张量，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）。

例子

```
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]) # 相当于 added = keras.layers.add([x1, x2])

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

5.8.2 Subtract [\[source\]](#)

`keras.layers.Subtract()`

计算两个输入张量的差。

相减层接受一个长度为 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。

例子

```
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)
```

5.8.3 Multiply [\[source\]](#)

`keras.layers.Multiply()`

计算一个列表的输入张量的（逐元素间的）乘积。

相乘层接受一个列表的张量，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）。

5.8.4 Average [\[source\]](#)

`keras.layers.Average()`

计算一个列表的输入张量的平均值。

平均层接受一个列表的张量，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）。

5.8.5 Maximum [\[source\]](#)

`keras.layers.Maximum()`

计算一个列表的输入张量的（逐元素间的）最大值。

最大层接受一个列表的张量，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）。

5.8.6 Concatenate [\[source\]](#)

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

串联一个列表的输入张量。

串联层接受一个列表的张量，除了串联轴之外，其他的尺寸都必须相同，然后返回一个由所有输入张量串联起来的输出张量。

参数

- **axis**: 串联的轴。
- ****kwargs**: 层关键字参数。

5.8.7 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。

5.8.8 add

`keras.layers.add(inputs)`

Add 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__**: 层关键字参数。

返回

一个张量，所有输入张量的和。

例子

```
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)
```

5.8.9 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。

例子

```
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)
```

5.8.10 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。

5.8.11 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。

5.8.12 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。

5.8.13 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。

5.8.14 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。

5.9 高级激活层 Advanced Activations

5.9.1 LeakyReLU [\[source\]](#)

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

带泄漏的修正线性单元。

当神经元未激活时，它仍可以赋予其一个很小的梯度： $f(x) = \alpha * x$ for $x < 0$, $f(x) = x$ 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 ≥ 0 。负斜率系数。

参考文献

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

5.9.2 PReLU [\[source\]](#)

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

参数化的修正线性单元。

形式： $f(x) = \alpha * x$ for $x < 0$, $f(x) = x$ for $x \geq 0$, 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](#)

5.9.3 ELU [\[source\]](#)

```
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 $x \geq 0$.

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 \(ELUs\)](#)

5.9.4 ThresholdedReLU [\[source\]](#)

```
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 ≥ 0 。激活的阈值位。

参考文献

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

5.9.5 Softmax [\[source\]](#)

```
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。

5.9.6 ReLU[source]

```
keras.layers.ReLU(max_value=None)
```

ReLU 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

- **max_value**: 浮点数，最大的输出值。

5.10 标准化层 Normalization

5.10.1 BatchNormalization [\[source\]](#)

```
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](#)

5.11 噪声层 Noise

5.11.1 GaussianNoise [\[source\]](#)

`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。

5.11.2 GaussianDropout [\[source\]](#)

`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 `sqrt(rate / (1 - 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

5.11.3 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，已在 dropout 之后仍然保证数据的自规范性。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 $\sqrt{\text{rate} / (1 - \text{rate})}$ 。
- **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](#)

5.12 层封装器 wrappers

5.12.1 TimeDistributed [\[source\]](#)

```
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。

5.12.2 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。

异常

- **ValueError**: 如果参数 `merge_mode` 非法。

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')
```


5.13 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 Keras2.0 中，Keras 层的骨架（如果你用的是旧的版本，请你更新）。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
import numpy as np

class MyLayer(Layer):

    def __init__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(MyLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        # Create a trainable weight variable for this layer.
        self.kernel = self.add_weight(name='kernel',
                                       shape=(input_shape[1], self.output_dim),
                                       initializer='uniform',
                                       trainable=True)
        super(MyLayer, self).build(input_shape) # Be sure to call this somewhere!

    def call(self, x):
        return K.dot(x, self.kernel)

    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.output_dim)
```

已有的 Keras 层就是实现层的很好例子。不要犹豫阅读源码！

6 数据预处理

6.1 序列预处理

6.1.1 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。

例子

```
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]]))

```

6.1.2 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 `padding` 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。
- **padding**: 字符串，‘pre’ 或 ‘post’，表示长度不足时是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 $(\text{len}(\text{sequences}), \text{maxlen})$ 。

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。

6.1.3 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 `wi` 的窗口是 $[i - \text{window_size}, i + \text{window_size} + 1]$ 。
- `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 $[0, 1, 1 \dots]$ ），如果 `True`，标签将为分类，例如 $[[1, 0], [0, 1], [0, 1] \dots]$ 。
- `sampling_table`: 尺寸为 `vocabulary_size` 的 1D 数组，其中第 i 项编码了排名为 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。

6.1.4 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，出现更频繁的单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：

$$p(\text{word}) = (\min(1, \sqrt{\text{word_frequency} / \text{sampling_factor}}) / (\text{word_frequency} / \text{sampling_factor}))$$

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 γ 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

A 1D Numpy array of length `size` where the *i*th entry is the probability that a word of rank *i* should be sampled.

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。

6.2 文本预处理

6.2.1 Tokenizer

```
keras.preprocessing.text.Tokenizer(num_words=None,
                                    filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~',
                                    lower=True, split=' ',
                                    char_level=False,
                                    oov_token=None)
```

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** 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 是会被分配给任何单词的保留索引。

6.2.2 hashing_trick

```
keras.preprocessing.text.hashing_trick(text, n,
                                       hash_function=None,
                                       filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~',
                                       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：`!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~`，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。

6.2.3 one_hot

```
keras.preprocessing.text.one_hot(text, n,
                                  filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~',
                                  lower=True, split=' ')
```

One-hot 将文本编码为大小为 n 的单词索引列表。

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：`!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~`，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）。

6.2.4 text_to_word_sequence

```
keras.preprocessing.text.text_to_word_sequence(text,
                                                  filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~',
                                                  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：`!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~`，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。

6.3 图像预处理

6.3.1 ImageDataGenerator 类

```
keras.preprocessing.image.ImageDataGenerator(featurewise_center=False,
                                              samplewise_center=False,
                                              featurewise_std_normalization=False,
                                              samplewise_std_normalization=False,
                                              zca_whitening=False,
                                              zca_epsilon=1e-06,
                                              rotation_range=0.0,
                                              width_shift_range=0.0,
                                              height_shift_range=0.0,
                                              brightness_range=None,
                                              shear_range=0.0,
                                              zoom_range=0.0,
                                              channel_shift_range=0.0,
                                              fill_mode='nearest',
                                              cval=0.0,
                                              horizontal_flip=False,
                                              vertical_flip=False,
                                              rescale=None,
                                              preprocessing_function=None,
                                              data_format=None,
                                              validation_split=0.0)
```

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。
- **`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”`: `aaaaaaaa|abcd|dddddddd`
 - `“reflect”`: `abddcba|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)。

例子

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)
```

6.3.2 ImageDataGenerator 类方法

6.3.2.1 apply_transform

```
keras.preprocessing.image.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）。

6.3.2.2 fit

```
keras.preprocessing.image.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。
- **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。

6.3.2.3 flow

```
keras.preprocessing.image.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 数组或一系列 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，而对于 RGB 数据，其值应该为 3。
- **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。

6.3.2.4 flow_from_directory

```
keras.preprocessing.image.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**: 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” 之一。默认: “rgb”。图像是否被转换成 1 或 3 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**: 如果目标尺寸与加载图像的尺寸不同，则使用插值方法重新采样图像。支持的方法有“nearest”，“bilinear”，and “bicubic”。如果安装了 1.1.3 以上版本的 PIL，还支持“lanczos”。如果安装了 3.4.0 以上版本的 PIL，还支持“box”和“hamming”。默认使用“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。

6.3.2.5 get_random_transform

```
keras.preprocessing.image.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。

6.3.2.6 random_transform

```
keras.preprocessing.image.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）。

6.3.2.7 standardize

`keras.preprocessing.image.standardize(x)`

将标准化配置应用于一批输入。

参数

- **x**: 需要标准化的一批输入。

返回

标准化后的输入。

7 损失函数 Losses

7.1 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](#)。

7.2 可用损失函数

7.2.1 mean_squared_error

```
mean_squared_error(y_true, y_pred)
```

7.2.2 mean_absolute_error

```
mean_absolute_error(y_true, y_pred)
```

7.2.3 mean_absolute_percentage_error

```
mean_absolute_percentage_error(y_true, y_pred)
```

7.2.4 mean_squared_logarithmic_error

```
mean_squared_logarithmic_error(y_true, y_pred)
```

7.2.5 squared_hinge

```
squared_hinge(y_true, y_pred)
```

7.2.6 hinge

```
hinge(y_true, y_pred)
```

7.2.7 categorical_hinge

`categorical_hinge(y_true, y_pred)`

7.2.8 logcosh

`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 $\text{abs}(x) - \log(2)$ 。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。

Arguments

- **y_true**: 目标真实值的张量。
- **y_pred**: 目标预测值的张量。

Returns

每个样本都有一个标量损失的张量。

7.2.9 categorical_crossentropy

`categorical_crossentropy(y_true, y_pred)`

7.2.10 sparse_categorical_crossentropy

`sparse_categorical_crossentropy(y_true, y_pred)`

7.2.11 binary_crossentropy

`binary_crossentropy(y_true, y_pred)`

7.2.12 kullback_leibler_divergence

`kullback_leibler_divergence(y_true, y_pred)`

7.2.13 poisson

`poisson(y_true, y_pred)`

7.2.14 cosine_proximity

`cosine_proximity(y_true, y_pred)`

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)
```

8 评估标准 Metrics

8.1 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](#)）。

8.1.1 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。

8.1.2 返回值

返回一个表示全部数据点平均值的张量。

8.2 可使用的评价函数

8.2.1 binary_accuracy

```
binary_accuracy(y_true, y_pred)
```

8.2.2 categorical_accuracy

```
categorical_accuracy(y_true, y_pred)
```

8.2.3 sparse_categorical_accuracy

```
sparse_categorical_accuracy(y_true, y_pred)
```

8.2.4 top_k_categorical_accuracy

```
top_k_categorical_accuracy(y_true, y_pred, k=5)
```

8.2.5 `sparse_top_k_categorical_accuracy`

```
sparse_top_k_categorical_accuracy(y_true, y_pred, k=5)
```

8.3 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])
```

9 优化器 Optimizers

9.1 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('tanh'))
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')
```

9.2 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)
```

9.2.1 SGD [\[source\]](#)

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=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

- **lr**: float >= 0. 学习率
- **momentum**: float >= 0. 参数, 用于加速 SGD 在相关方向上前进, 并抑制震荡
- **decay**: float >= 0. 每次参数更新后学习率衰减值.
- **nesterov**: boolean. 是否使用 Nesterov 动量.

9.2.2 RMSprop [\[source\]](#)

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
```

RMSProp 优化器.

建议使用优化器的默认参数 (除了学习率 lr, 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

- **lr**: float >= 0. 学习率.
- **rho**: float >= 0. RMSProp 梯度平方的移动均值的衰减率.
- **epsilon**: float >= 0. 模糊因子. 若为 None, 默认为 K.epsilon().
- **decay**: float >= 0. 每次参数更新后学习率衰减值.

引用

- [rmsprop: Divide the gradient by a running average of its recent magnitude](#)

9.2.3 Adagrad [\[source\]](#)

```
keras.optimizers.Adagrad(lr=0.01, epsilon=None, decay=0.0)
```

Adagrad 优化器.

建议使用优化器的默认参数。

参数

- **lr**: float >= 0. 学习率.
- **epsilon**: float >= 0. 若为 None, 默认为 K.epsilon().
- **decay**: float >= 0. 每次参数更新后学习率衰减值.

引用

- [Adaptive Subgradient Methods for Online Learning and Stochastic Optimization](#)

9.2.4 Adadelta [\[source\]](#)

```
keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=None, decay=0.0)
```

Adagrad 优化器。

建议使用优化器的默认参数。

参数

- **lr**: float ≥ 0 . 学习率, 建议保留默认值.
- **rho**: float ≥ 0 . Adadelta 梯度平方移动均值的衰减率
- **epsilon**: float ≥ 0 . 模糊因子. 若为 `None`, 默认为 `K.epsilon()`.
- **decay**: float ≥ 0 . 每次参数更新后学习率衰减值.

引用

- [Adadelta - an adaptive learning rate method](#)

9.2.5 Adam [\[source\]](#)

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None,  
                        decay=0.0, amsgrad=False)
```

Adam 优化器。

默认参数遵循原论文中提供的值。

参数

- **lr**: float ≥ 0 . 学习率.
- **beta_1**: float, $0 < \beta_1 < 1$. 通常接近于 1.
- **beta_2**: float, $0 < \beta_2 < 1$. 通常接近于 1.
- **epsilon**: float ≥ 0 . 模糊因子. 若为 `None`, 默认为 `K.epsilon()`.
- **decay**: float ≥ 0 . 每次参数更新后学习率衰减值.
- **amsgrad**: boolean. 是否应用此算法的 AMSGrad 变种, 来自论文"On the Convergence of Adam and Beyond".

引用

- [Adam - A Method for Stochastic Optimization](#)
- [On the Convergence of Adam and Beyond](#)

9.2.6 Adamax [\[source\]](#)

```
keras.optimizers.Adamax(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0)
```

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

- **lr**: float ≥ 0 . 学习率.
- **beta_1/beta_2**: floats, $0 < \text{beta} < 1$. 通常接近于 1.
- **epsilon**: float ≥ 0 . 模糊因子. 若为 None, 默认为 `K.epsilon()`.
- **decay**: float ≥ 0 . 每次参数更新后学习率衰减.

引用

- [Adam - A Method for Stochastic Optimization](#)

9.2.7 Nadam [\[source\]](#)

```
keras.optimizers.Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=None,  
                        schedule_decay=0.004)
```

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

- **lr**: float ≥ 0 . 学习率.
- **beta_1/beta_2**: floats, $0 < \text{beta} < 1$. 通常接近于 1.
- **epsilon**: float ≥ 0 . 模糊因子. 若为 None, 默认为 `K.epsilon()`.

引用

- [Nadam report](#)
- [On the importance of initialization and momentum in deep learning](#)

9.2.8 TFOptimizer [\[source\]](#)

```
keras.optimizers.TFOptimizer(optimizer)
```

原生 Tensorflow 优化器的包装类 (wrapper class)。

10 激活函数 Activations

10.1 激活函数的用法

激活函数可以通过设置单独的激活层实现，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))
```

10.2 预定义激活函数

10.2.1 softmax

```
softmax(x, axis=-1)
```

Softmax 激活函数.

Arguments

`x`: 张量. - `axis`: 整数, 代表 softmax 所作用的维度

Returns

softmax 变换后的张量.

Raises

- **ValueError**: In case `dim(x) == 1`.

10.2.2 elu

```
elu(x, alpha=1.0)
```

10.2.3 selu

`selu(x)`

可伸缩的指数线性单元 (Klambauer et al., 2017)。

Arguments

- **x**: 一个用来用于计算激活函数的张量或变量。

Returns

与 **x** 具有相同类型及形状的张量。

Note

- 与“lecun_normal”初始化方法一起使用。
- 与 dropout 的变种“AlphaDropout”一起使用。

References

- [Self-Normalizing Neural Networks](#)

10.2.4 softplus

`softplus(x)`

10.2.5 softsign

`softsign(x)`

10.2.6 relu

`relu(x, alpha=0.0, max_value=None)`

10.2.7 tanh

`tanh(x)`

10.2.8 sigmoid

`sigmoid(x)`

10.2.9 hard_sigmoid

`hard_sigmoid(x)`

10.2.10 linear

`linear(x)`

10.3 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，如 PReLU, LeakyReLU 等

11 回调函数 Callbacks

11.1 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。

11.1.1 Callback [\[source\]](#)

```
keras.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）。

11.1.2 BaseLogger [\[source\]](#)

```
keras.callbacks.BaseLogger()
```

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。

11.1.3 TerminateOnNaN [\[source\]](#)

```
keras.callbacks.TerminateOnNaN()
```

当遇到 NaN 损失会停止训练的回调函数。

11.1.4 ProgbarLogger [\[source\]](#)

```
keras.callbacks.ProgbarLogger(count_mode='samples')
```

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）。

触发

- **ValueError**: 防止不正确的 count_mode

11.1.5 History [\[source\]](#)

```
keras.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。

11.1.6 ModelCheckpoint [\[source\]](#)

```
keras.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）。

11.1.7 EarlyStopping [\[source\]](#)

```
keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=0,  
                               verbose=0, mode='auto')
```

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**: 没有进步的训练轮数，在这之后训练就会被停止。
- **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。

11.1.8 RemoteMonitor [\[source\]](#)

```
keras.callbacks.RemoteMonitor(root='http://localhost:9000',  
                               path='/publish/epoch/end/', field='data', headers=None)
```

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。

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。

11.1.9 LearningRateScheduler [\[source\]](#)

```
keras.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。

11.1.10 TensorBoard [\[source\]](#)

```
keras.callbacks.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)
```

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
```

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。
- **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。
- **batch_size**: 用以直方图计算的传入神经网络输入批的大小。
- **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）。
- **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称。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。

11.1.11 ReduceLROnPlateau [\[source\]](#)

```
keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=10,
                                   verbose=0, mode='auto', epsilon=0.0001, cooldown=0, min_lr=0)
```

当标准评估已经停止时，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

```
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**: 没有进步的训练轮数，在这之后训练速率会被降低。
- **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，学习速率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。
- **epsilon**: 对于测量新的最优化的阈值，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。

11.1.12 CSVLogger [\[source\]](#)

```
keras.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

```
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。

11.1.13 LambdaCallback [\[source\]](#)

```
keras.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。

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, 但是时每一行都是 *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])
```

11.2 创建一个回调函数

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'))
```

11.2.1 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]
'''
```

11.2.2 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])
```

12 常用数据集 Datasets

12.1 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)。
 - **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,)。

12.2 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)。
 - **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,)。
- 参数：
- **label_mode**: "fine" 或者 "coarse"

12.3 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。每一条评论已经过预处理，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


```
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。

12.5 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。

12.6 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
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,)。

12.7 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)。

13 预训练模型 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，会自动下载权值到 `~/.keras/models/` 目录下。

13.1 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
- EesNet50
- InceptionV3
- InceptionResNetV2
- MobileNet
- DenseNet
- NASnet

所有的这些模型（除了 Xception 和 MobileNet 外）都兼容 Theano 和 Tensorflow，并会自动按照位于 `~/.keras/keras.json` 的配置文件中设置的图像数据格式来构建模型。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。

Xception 模型仅适用于 TensorFlow，因为它依赖于 SeparableConvolution 层。MobileNet 模型仅适用于 TensorFlow，因为它依赖于 DepthwiseConvolution 层。

13.2 图像分类模型的示例代码

13.2.1 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)
# decode the results into a list of tuples (class, description, probability)
# (one such list for each sample in the batch)
print('Predicted:', decode_predictions(preds, top=3)[0])
# Predicted: [(u'n02504013', u'Indian_elephant', 0.82658225), (u'n01871265', u'tusker', 0.1
```

13.2.2 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)
```

13.2.3 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)
```

13.2.4 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(...)
```

13.2.5 通过自定义输入 tensor 构建 InceptionV3

```
from keras.applications.inception_v3 import InceptionV3
from keras.layers import Input

# this could also be the output a different Keras model or layer
input_tensor = Input(shape=(224, 224, 3)) # this assumes K.image_data_format() == 'channel.

model = InceptionV3(input_tensor=input_tensor, weights='imagenet', include_top=True)
```

13.3 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	0.715	0.901	138,357,544	23
VGG19	549 MB	0.727	0.910	143,667,240	26
EesNet50	99 MB	0.759	0.929	25,636,712	168
InceptionV3	92 MB	0.788	0.944	23,851,784	159
InceptionResNetV2	215 MB	0.804	0.953	55,873,736	572
MobileNet	17 MB	0.665	0.871	4,253,864	88
DenseNet121	33 MB	0.745	0.918	8,062,504	121
DenseNet169	57 MB	0.759	0.928	14,307,880	169
DenseNet201	80 MB	0.770	0.933	20,242,984	201

Top-1 准确率和 Top-5 准确率都是在 ImageNet 验证集上的结果。

13.3.1 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。

注意，该模型目前仅能在 TensorFlow 后端使用，因为它依赖 SeparableConvolution 层，目前该层只支持 channels_last 的维度顺序（高度、宽度、通道）。

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（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 (299, 299, 3)，因为预训练模型是以这个大小训练的）。输入尺寸必须是三个数字，且宽高必须不小于 71，比如 (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层的输出，该输出是一个四维张量。
 - 'avg' 代表全局平均池化（GlobalAveragePool2D），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。

返回值

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。

13.3.2 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。

该模型在 Theano 和 TensorFlow 后端均可使用，并接受 `channels_first` 和 `channels_last` 两种输入维度顺序（高度，宽度，通道）。

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（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）。输入尺寸必须是三个数字，且宽高必须不小于 48，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（`GlobalAveragePool2D`），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。

返回值

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](#)。

13.3.3 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。

该模型在 Theano 和 TensorFlow 后端均可使用, 并接受 `channels_first` 和 `channels_last` 两种输入维度顺序 (高度, 宽度, 通道)。

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 (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)。输入尺寸必须是三个数字, 且宽高必须不小于 48, 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 (`GlobalAveragePool2D`), 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。

返回值

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](#)。

13.3.4 ResNet50

```
keras.applications.resnet50.ResNet50(include_top=True, weights='imagenet',  
                                     input_tensor=None, input_shape=None, pooling=None, classes=1000)
```

ResNet50 模型，权值由 ImageNet 训练而来。

该模型在 Theano 和 TensorFlow 后端均可使用，并接受 `channels_first` 和 `channels_last` 两种输入维度顺序（高度，宽度，通道）。

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（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）。输入尺寸必须是三个数字，且宽高必须不小于 197，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（`GlobalAveragePool2D`），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。

返回值

一个 Keras Model 对象。

参考文献

- [Deep Residual Learning for Image Recognition](#)

License

预训练权值由 [Kaiming He](#) 发布的预训练权值移植而来，基于 [MIT license](#)。

13.3.5 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。

该模型在 Theano 和 TensorFlow 后端均可使用，并接受 `channels_first` 和 `channels_last` 两种输入维度顺序（高度，宽度，通道）。

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 (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 (299, 299, 3) (`channels_last` 格式) 或 (3, 299, 299) (`channels_first` 格式), 因为预训练模型是以这个大小训练的)。输入尺寸必须是三个数字, 且宽高必须不小于 139, 比如 (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层的输出, 该输出是一个四维张量。
 - `'avg'` 代表全局平均池化 (`GlobalAveragePool2D`), 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。

返回值

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](#)。

13.3.6 InceptionResNetV2

```
keras.applications.inception_resnet_v2.InceptionResNetV2(include_top=True, weights='imagenet',  
input_shape=None, pooling=None, classes=1000)
```

Inception-ResNet V2 模型, 权值由 ImageNet 训练而来。

该模型在 Theano 和 TensorFlow 后端均可使用, 并接受 `channels_first` 和 `channels_last` 两种输入维度顺序 (高度, 宽度, 通道)。

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 (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 (299, 299, 3)（`channels_last` 格式）或 (3, 299, 299)（`channels_first` 格式），因为预训练模型是以这个大小训练的）。输入尺寸必须是三个数字，且宽高必须不小于 139，比如 (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层的输出，该输出是一个四维张量。
 - `'avg'` 代表全局平均池化（`GlobalAveragePool2D`），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。

返回值

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](#)。

13.3.7 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。

注意，该模型目前仅能在 TensorFlow 后端使用，因为它依赖 `SeparableConvolution` 层，目前该层只支持 `channels_last` 的维度顺序（高度、宽度、通道）。

要通过 `load_model` 载入 MobileNet 模型，你需要导入自定义对象 `relu6` 和 `DepthwiseConv2D` 并通过 `custom_objects` 传参。

下面是示例代码：

```
model = load_model('mobilenet.h5', custom_objects={
    'relu6': mobilenet.relu6,
    'DepthwiseConv2D': mobilenet.DepthwiseConv2D})
```

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 (不然输入形状必须是 (224, 224, 3), 因为预训练模型是以这个大小训练的)。输入尺寸必须是三个数字, 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层的输出, 该输出是一个四维张量。
 - `'avg'` 代表全局平均池化 (`GlobalAveragePool2D`), 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

- [MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications](#)

License

预训练权值基于 [Apache License](#)。

13.3.8 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)
```

可以选择载入在 ImageNet 上的预训练权值。如果你在使用 TensorFlow 为了发挥最佳性能, 请在 `~/.keras/keras.json` 的 Keras 配置文件中设置 `image_data_format='channels_last'`。

模型和权值兼容 TensorFlow、Theano 和 CNTK。可以在你的 Keras 配置文件中指定数据格式。

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)。输入尺寸必须是三个数字。
- `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 (`GlobalAveragePool2D`), 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。

A Keras model instance.

返回

一个 Keras Model 对象。

参考文献

预训练权值基于 [BSD 3-clause License](#)。

13.3.9 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)。

注意, 该模型目前仅能在 TensorFlow 后端使用, 因此它只支持 `channels_last` 的维度顺序 (高度、宽度、通道), 可以在 `~/.keras/keras.json` Keras 配置文件中设置。

NASNetLarge 默认输入尺寸是 331x331, NASNetMobile 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 (不然对于 NASNetMobile 模型来说, 输入形状必须是 (224, 224, 3) (`channels_last` 格式), 对于 NASNetLarge 来说, 输入形状必须是 (331, 331, 3) (`channels_last` 格式)。输入尺寸必须是三个数字。

- `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)。输入尺寸必须是三个数字。
- `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 (`GlobalAveragePool2D`), 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

- [Learning Transferable Architectures for Scalable Image Recognition](#)

License

预训练权值基于 [Apache License](#)。

14 后端 Backend

14.1 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。

14.2 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.
```

14.3 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"。

14.4 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
```

14.5 后端函数

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。

例子

```
>>> keras.backend.epsilon()
1e-07
```

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。

例子

```
>>> 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。

例子

```
>>> 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'。

例子

```
>>> 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。

例子

```
>>> 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 ('channels_first' 或 'channels_last')。

返回

一个字符串, 'channels_first' 或 'channels_last'

例子

```
>>> 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'。

例子

```
>>> from keras import backend as K
>>> K.image_data_format()
'channels_first'
>>> K.set_image_data_format('channels_last')
>>> K.image_data_format()
'channels_last'
```

get_uid

```
keras.backend.get_uid(prefix='')
```

获取默认计算图的 uid。

参数

- **prefix**: 图的可选前缀。

返回

图的唯一标识符。

reset_uids

```
keras.backend.reset_uids()
```

重置图的标识符。

clear_session

```
keras.backend.clear_session()
```

销毁当前的 TF 图并创建一个新图。

有助于避免旧模型/网络层混乱。

manual_variable_initialization

```
keras.backend.manual_variable_initialization(value)
```

设置变量手动初始化的标志。

这个布尔标志决定了变量是否应该在实例化时初始化（默认），或者用户是否应该自己处理初始化（例如通过 `tf.initialize_all_variables()`）。

参数

- **value**: Python 布尔值。

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。

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。

例子

```
>>> 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。

例子

```
>>> 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）

例子

```
>>> from keras import backend as K
>>> val = np.array([[1, 2], [3, 4]])
>>> kvar = K.variable(value=val, dtype='float64', name='example_var')
>>> K.dtype(kvar)
'float64'
>>> print(kvar)
example_var
>>> K.eval(kvar)
array([[ 1.,  2.],
       [ 3.,  4.]])
```

constant

`keras.backend.constant(value, dtype=None, shape=None, name=None)`

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。

例子

```
>>> 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))
>>> K.is_keras_tensor(k_var) # 在 Keras 之外间接创建的变量不是 Keras 张量。
False
>>> keras_var = K.variable(np_var)
>>> K.is_keras_tensor(keras_var) # Keras 后端创建的变量不是 Keras 张量。
False
>>> keras_placeholder = K.placeholder(shape=(2, 4, 5))
>>> K.is_keras_tensor(keras_placeholder) # 占位符不是 Keras 张量。
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)
>>> K.is_keras_tensor(keras_layer_output) # 任何 Keras 层输出都是 Keras 张量。
True
```

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）。

例子

```
>>> 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）。

例子

TensorFlow 例子

```
>>> 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)。

例子

```
>>> from keras import backend as K
>>> inputs = K.placeholder(shape=(2, 4, 5))
>>> K.int_shape(inputs)
(2, 4, 5)
>>> val = np.array([[1, 2], [3, 4]])
>>> kvar = K.variable(value=val)
>>> K.int_shape(kvar)
(2, 2)
```

ndim

keras.backend.ndim(x)

以整数形式返回张量中的轴数。

参数

- x: 张量或变量。

返回

Integer (scalar), number of axes.

例子

```
>>> from keras import backend as K
>>> inputs = K.placeholder(shape=(2, 4, 5))
>>> val = np.array([[1, 2], [3, 4]])
>>> kvar = K.variable(value=val)
>>> K.ndim(inputs)
3
>>> K.ndim(kvar)
2
```

dtype

`keras.backend.dtype(x)`

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。

例子

```
>>> 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]]))
>>> K.dtype(kvar)
'float32_ref'
>>> kvar = K.variable(np.array([[1, 2], [3, 4]]), dtype='float32')
>>> K.dtype(kvar)
'float32_ref'
```

eval

```
keras.backend.eval(x)
```

估计一个变量的值。

参数

- **x**: 变量。

返回

Numpy 数组。

例子

```
>>> from keras import backend as K
>>> kvar = K.variable(np.array([[1, 2], [3, 4]]), dtype='float32')
>>> K.eval(kvar)
array([[ 1.,  2.],
       [ 3.,  4.]], dtype=float32)
```

zeros

```
keras.backend.zeros(shape, dtype=None, name=None)
```

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。

例子

```
>>> from keras import backend as K
>>> kvar = K.zeros((3,4))
>>> K.eval(kvar)
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]], dtype=float32)
```

ones

```
keras.backend.ones(shape, dtype=None, name=None)
```

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。

例子

```
>>> from keras import backend as K
>>> kvar = K.ones((3,4))
>>> K.eval(kvar)
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]], dtype=float32)
```

eye

```
keras.backend.eye(size, dtype=None, name=None)
```

实例化一个单位矩阵并返回它。

参数

- **size**: 整数，行/列的数目。
- **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。

例子

```
>>> from keras import backend as K
>>> kvar = K.eye(3)
>>> K.eval(kvar)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]], dtype=float32)
```

zeros_like

```
keras.backend.zeros_like(x, dtype=None, name=None)
```

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。

例子

```
>>> from keras import backend as K
>>> kvar = K.variable(np.random.random((2,3)))
>>> kvar_zeros = K.zeros_like(kvar)
>>> K.eval(kvar_zeros)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]], dtype=float32)
```

ones_like

```
keras.backend.ones_like(x, dtype=None, name=None)
```

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 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，用一填充。

例子

```
>>> from keras import backend as K
>>> kvar = K.variable(np.random.random((2,3)))
>>> kvar_ones = K.ones_like(kvar)
>>> K.eval(kvar_ones)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]], dtype=float32)
```

identity

```
keras.backend.identity(x, name=None)
```

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。

例子

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)
```

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。

例子

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)
```

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。

例子

```
>>> kvar = K.zeros((2,3))
>>> K.count_params(kvar)
6
>>> K.eval(kvar)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]], dtype=float32)
```

cast

`keras.backend.cast(x, dtype)`

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。

例子

```
>>> 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 $(2, 3) * (4, 3, 5) \rightarrow (2, 4, 5)$)

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。

例子

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* 行为的例子

```
>>> x = K.random_uniform_variable(shape=(2, 3), low=0, high=1)
>>> y = K.ones((4, 3, 5))
>>> xy = K.dot(x, y)
>>> K.int_shape(xy)
(2, 4, 5)
```

batch_dot

```
keras.backend.batch_dot(x, y, axes=None)
```

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`: 表示目标维度的整数或列表。`axes[0]` 和 `axes[1]` 的长度必须相同。

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）的连接张量。如果最后的秩为 1，我们将它重新转换为 $(batch_size, 1)$ 。

例子

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。

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)
```

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。

例子

```
>>> var = K.variable([[1, 2, 3], [4, 5, 6]])
>>> K.eval(var)
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]], dtype=float32)
>>> var_transposed = K.transpose(var)
>>> K.eval(var_transposed)
array([[ 1.,  4.],
       [ 2.,  5.],
       [ 3.,  6.]], dtype=float32)

>>> inputs = K.placeholder((2, 3))
>>> inputs
<tf.Tensor 'Placeholder_11:0' shape=(2, 3) dtype=float32>
>>> input_transposed = K.transpose(inputs)
>>> input_transposed
<tf.Tensor 'transpose_4:0' shape=(3, 2) dtype=float32>
```

gather

```
keras.backend.gather(reference, indices)
```

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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)。

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)。

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。

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。

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。

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 $\log(\text{sum}(\exp(\text{张量在某一轴的元素})))$ 。

这个函数在数值上比 $\log(\text{sum}(\exp(x)))$ 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。

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。

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 浮点或整数。
- **max_value**: Python 浮点或整数。

返回

一个张量。

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。

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。

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。

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。

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。

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。

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。

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。

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: $\text{output} = (x - \text{mean}) / (\sqrt{\text{var} + \text{epsilon}}) * \text{gamma} + \text{beta}$

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。

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。

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 $(batch_size, dim1, dim2, \dots dim(n-1))$
- **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 $(batch_size, dim1, dim2, \dots dim(n-1), num_classes)$ 。

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。

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**: 需要设置新值的张量。
- **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。

例子

```
>>> 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。

参数

- **inputs**: 占位符张量列表。
- **outputs**: 输出张量列表。
- **updates**: 更新操作列表。
- ****kwargs**: 需要传递给 `tf.Session.run` 的参数。

返回

输出值为 Numpy 数组。

异常

- **ValueError**: 如果无效的 kwargs 被传入。

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)。

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序。

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。

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。

softmax

```
keras.backend.softmax(x)
```

张量的 Softmax 值。

参数

- **x**: 张量或变量。

返回

一个张量。

softplus

```
keras.backend.softplus(x)
```

张量的 Softplus 值。

参数

- **x**: 张量或变量。

返回

一个张量。

softsign

`keras.backend.softsign(x)`

张量的 Softsign 值。

参数

- **x**: 张量或变量。

返回

一个张量。

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。

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。

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。

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。

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。

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"。

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:

1. 无效的 **data_format** 参数。
2. 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。

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)
```

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 false, 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,)。

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。

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`。

backend

```
backend.backend()
```

公开可用的方法，以确定当前后端。

返回

字符串，Keras 目前正在使用的后端名。

例子

```
>>> keras.backend.backend()  
'tensorflow'
```

15 初始化 Initializers

15.1 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'))
```

15.2 可用的初始化器

下面这些是可用的内置初始化器，是 `keras.initializers` 模块的一部分:

15.2.1 `Initializer` [\[source\]](#)

```
keras.initializers.Initializer()
```

初始化器基类：所有初始化器继承这个类。

15.2.2 `Zeros` [\[source\]](#)

```
keras.initializers.Zeros()
```

将张量初始值设为 0 的初始化器。

15.2.3 `Ones` [\[source\]](#)

```
keras.initializers.Ones()
```

将张量初始值设为 1 的初始化器。

15.2.4 `Constant` [\[source\]](#)

```
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。

15.2.5 RandomNormal [\[source\]](#)

```
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。

15.2.6 RandomUniform [\[source\]](#)

```
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上限。默认为浮点类型的 1。
- **seed**: 一个 Python 整数。用于设置随机数种子。

15.2.7 TruncatedNormal [\[source\]](#)

```
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。

15.2.8 VarianceScaling [\[source\]](#)

```
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， $\text{stddev} = \sqrt{\text{scale} / n}$ ，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 `[-limit, 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。

15.2.9 Orthogonal [\[source\]](#)

```
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

Saxe et al., <http://arxiv.org/abs/1312.6120>

15.2.10 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。

15.2.11 lecun_uniform

```
lecun_uniform(seed=None)
```

LeCun 均匀初始化器。

它从 `[-limit, limit]` 中的均匀分布中抽取样本，其中 `limit` 是 `sqrt(3 / fan_in)`，`fan_in` 是权值张量中的输入单位的数量。

Arguments

- **seed**: 一个 Python 整数。作为随机发生器的种子。

Returns

一个初始化器。

参考文献

LeCun 98, Efficient Backprop, - <http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf>

15.2.12 glorot_normal

`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 $\text{stddev} = \sqrt{2 / (\text{fan_in} + \text{fan_out})}$ 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。

Arguments

- **seed**: 一个 Python 整数。作为随机发生器的种子。

Returns

一个初始化器。

参考文献

Glorot & Bengio, AISTATS 2010 - <http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf>

15.2.13 glorot_uniform

`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 `[-limit, 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

Glorot & Bengio, AISTATS 2010 - <http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf>

15.2.14 he_normal

`he_normal(seed=None)`

He 正态分布初始化器。

它从以 0 为中心，标准差为 $\text{stddev} = \sqrt{2 / \text{fan_in}}$ 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

He et al., <http://arxiv.org/abs/1502.01852>

15.2.15 lecun_normal

`lecun_normal(seed=None)`

LeCun 正态分布初始化器。

它从以 0 为中心，标准差为 $\text{stddev} = \sqrt{1 / \text{fan_in}}$ 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](#)

15.2.16 he_uniform

`he_uniform(seed=None)`

He 均匀方差缩放初始化器。

它从 `[-limit, 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

He et al., <http://arxiv.org/abs/1502.01852>

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'))
```

15.3 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))
```

16 正则化 Regularizers

16.1 正规化的使用

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

16.2 例子

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01),
                activity_regularizer=regularizers.l1(0.01)))
```

16.3 可用的惩罚

```
keras.regularizers.l1(0.)
keras.regularizers.l2(0.)
keras.regularizers.l1_l2(0.)
```

16.4 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写你的正则化器，例子见 <https://github.com/keras-team/keras/blob/master/keras/regularizers.py> 模块。

17 约束 Constraints

17.1 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.)))
```

17.2 可用的约束

- `max_norm(max_value=2, axis=0)`: 最大范数约束
- `non_neg()`: 非负性约束
- `unit_norm(axis=0)`: 单位范数约束
- `min_max_norm(min_value=0.0, max_value=1.0, rate=1.0, axis=0)`: 最小/最大范数约束

18 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` 有两个可选参数：

- `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。

此外，你也可以直接取得 `pydot.Graph` 对象并自己渲染它。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'))
```

19 Scikit-learn API

Scikit-Learn API 的封装器

你可以使用 Keras 的顺序模型 (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 regressor 界面。

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 这应该建立, 编译, 并返回一个 Keras 模型, 将被用来拟合/预测。以下之一三个值可以传递给 **build_fn**

1. 函数
2. 实现 `__call__` 函数的类的实例
3. 没有。这意味着你实现了一个继承的类 `KerasClassifier` 或 `KerasRegressor`。当前类的 `__call__` 函数将被视为默认的 **build_fn**。

sk_params 同时包含模型参数和拟合参数。法律模型参数是 **build_fn** 的参数。类似于其他所有估计者在 Scikit-Learn, **build_fn** 应该为其参数提供默认值, 这样就可以创建估计器, 而不需要将任何值传递给 **sk_params**。

sk_params 也可以被称为 **fit**, **predict**, **predict_proba** 和 **score** 函数 (e.g., **epochs**, **batch_size**). 拟合 (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` 的默认值 **fit**, **predict**, **predict_proba** 和 **score** 函数

当 scikit-learn 使用 **grid_search** API 时, 有效参数与 **sk_params** 相同, 包括拟合参数。换句话说, 你可以使用 **grid_search** 来搜索最好的 **batch_size** 或 **epochs** 以及模型参数。

20 工具

20.1 CustomObjectScope [\[source\]](#)

`keras.utils.CustomObjectScope()`

提供一个无法转义的 `_GLOBAL_CUSTOM_OBJECTS` 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。

例子

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
```

20.2 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。

例子

```
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

一个类数组的 HDF5 数据集。

20.3 Sequence [\[source\]](#)

`keras.utils.Sequence()`

用于拟合数据序列的基类，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。

例子

```
from skimage.io import imread
from skimage.transform import resize
import numpy as np
import math

# 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 math.ceil(len(self.x) / 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)
```

20.4 to_categorical

```
keras.utils.to_categorical(y, num_classes=None)
```

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。

返回

输入的二进制矩阵表示。

20.5 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。

20.6 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。

20.7 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)。

20.8 plot_model

```
keras.utils.plot_model(model, to_file='model.png', show_shapes=False,
                        show_layer_names=True, rankdir='TB')
```

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。

20.9 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 ≥ 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。

例子

```
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「模板」模型）。
# 我们建议在 CPU 设备范围内执行此操作，
# 以便让模型的权值托管在 CPU 内存上。
# 否则，他们可能会最终托管在 GPU 上，
# 这会使权值共享变得复杂。
with tf.device('/cpu:0'):
    model = Xception(weights=None,
                      input_shape=(height, width, 3),
                      classes=num_classes)
```

```
# 将模型复制到 8 个 GPU 上。
# 这假定你的机器有 8 个可用的 GPU。
parallel_model = multi_gpu_model(model, gpus=8)
parallel_model.compile(loss='categorical_crossentropy',
                      optimizer='rmsprop')

# 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 批大小为 256, 每个 GPU 将处理 32 个样本。
parallel_model.fit(x, y, epochs=20, batch_size=256)

# 通过模板模型（共享相同的权值）保存模型：
model.save('my_model.h5')
```

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。

21 贡献

21.1 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。

21.2 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。

21.3 请求新功能

你也可以使用 Github Issue 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，我们需要的功能是针对大多数用户而言的，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. 讨论完该功能后，您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。

21.4 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。

21.5 Pull Requests 合并请求

我应该在哪儿提交我的合并请求？

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 [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。

21.6 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。

♥ 更多科研资源，请扫码关注微信公众号：



♥ 如诸君认为本文档对您的使用和研究略有帮助，不妨扫码请笔者喝杯咖啡，以鼓励作者进一步完善文档内容，提高文档质量。☺



微信支付