Keras 中文文档

Content in Detailed

- Keras:基于 Python 的深度学习库
    - 停止更新通知 @2017（如有什么想交流的仍然可以通过 moyan_work@foxmail.com 联系我，同时也可以加我们的 Keras 群 119427073 或 523412399 一起聊天吹水）
    - 这就是 Keras modularity, extensible, etc.
    - 关于 Keras-cn 本文档的额外模块还有：Keras 新手指南,Keras 资源,深度学习与 Keras.
    - 当前版本与更新 目前文档的版本号是 2.0.9，对应于官方的 2.0.9
    - 快速开始：30s 上手 Keras

```
from keras.models import Sequential
model = Sequential()
from keras.layers import Dense, Activation

model.add(Dense(units=64, input_dim=100))
model.add(Activation("relu"))
model.add(Dense(units=10))
model.add(Activation("softmax"))

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

from keras.optimizers import SGD

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

odel.fit(x_train, y_train, epochs=5, batch_size=32)

model.train_on_batch(x_batch, y_batch)

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

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

为了更深入的了解 Keras，我们建议你查看一下下面的两个 tutorial

- 快速开始 Sequntial 模型
- 快速开始函数式模型
    - 安装
    - 在 Theano、CNTK、TensorFlow 间切换
      技术支持
    - Keras Google group
- Keras Slack channel,点击这里获得邀请.

你也可以在 Github issues 里提问或请求新特性。对于习惯中文的用户，我们推荐在“集智”平台提问，该平台由 Kaiser 等搭建，支持在线代码运行环境，我本人会经常访问该网站解答问题

最后，我们也欢迎同学们加我们的 QQ 群 119427073 进行讨论（潜水和灌水会被 T，入群说明公司/学校-职位/年级）

-
- 致谢
- Keras 后端
    - 什么是“后端”
    - Keras 是一个模型级的库，提供了快速构建深度学习网络的模块。Keras 并不处理如张量乘法、卷积等底层操作。这些操作依赖于某种特定的、优化良好的张量操作库。Keras 依赖于处理张量的库就称为“后端引擎”。Keras 提供了三种后端引擎 Theano/Tensorflow/CNTK，并将其函数统一封装，使得用户可以以同一个接口调用不同后端引擎的函数
    - 切换后端 `$HOME/.keras/keras.json`
    - 我们也可以通过定义环境变量 `KERAS_BACKEND` 来覆盖上面配置文件中定义的后端：
    - `KERAS_BACKEND=tensorflow python -c "from keras import backend;" Using TensorFlow backend.`
    - keras.json 细节 你可以更改以上`~/.keras/keras.json` 中的配置
- `iamge_data_format`：字符串，"channels_last"或"channels_first"，该选项指定了 Keras 将要使用的维度顺序，可通过 `keras.backend.image_data_format()`来获取当前的维度顺序。对 2D 数据来说，"channels_last"假定维度顺序为(rows,cols,channels)而"channels_first"假定维度顺序为(channels, rows, cols)。对 3D 数据而言，"channels_last"假定(conv_dim1, conv_dim2, conv_dim3, channels)，"channels_first"则是(channels, conv_dim1, conv_dim2, conv_dim3)
- `epsilon`：浮点数，防止除 0 错误的小数字
- `floatx`：字符串，`"float16"`, `"float32"`, `"float64"`之一，为浮点数精度
- `backend`：字符串，所使用的后端，为"tensorflow"或"theano"
    - 使用抽象的 Keras 后端来编写代码 如果你希望你编写的 Keras 模块能够同时在 Theano 和 TensorFlow 两个后端上使用，你可以通过 Keras 后端接口来编写代码，这里是一个简介：`from keras import backend as K`
    - 下面的代码实例化了一个输入占位符，等价于 `tf.placeholder()`，`T.matrix()`，`T.tensor3()`等
    - `input = K.placeholder(shape=(2, 4, 5))`

```
# also works:
input = K.placeholder(shape=(None, 4, 5))
# also works:
input = K.placeholder(ndim=3)
```

    - 下面的代码实例化了一个共享变量（shared），等价于 `tf.variable()`或 `theano.shared()`
    - `val = np.random.random((3, 4, 5))`

```
var = K.variable(value=val)
```

```
# all-zeros variable:
var = K.zeros(shape=(3, 4, 5))
# all-ones:
var = K.ones(shape=(3, 4, 5))
```

- 大多数你需要的张量操作都可以通过统一的 Keras 后端接口完成，而不关心具体执行这些操作的是 Theano 还是 TensorFlow
- a = b + c * K.abs(d)

```
c = K.dot(a, K.transpose(b))
a = K.sum(b, axis=2)
a = K.softmax(b)
a = concatenate([b, c], axis=-1)
# etc…
```

- Kera 后端函数 len(dir(keras)) → 32
- backend()
- >>> keras.backend.backend()

```
'tensorflow'
```

- epsilon()以数值形式返回一个（一般来说很小的）数，用以防止除 0 错误 set_epsilon(e)设置在数值表达式中使用的 fuzz factor，用于防止除 0 错误，该值应该是一个较小的浮点数，示例：>>> from keras import backend as K

```
>>> K.epsilon()
1e-08
>>> K.set_epsilon(1e-05)
>>> K.epsilon()
1e-05
```

- floatx()返回默认的浮点数数据类型，为字符串，如 'float16', 'float32', 'float64'
- **set_floatx(floatx)设置默认的浮点数数据类型，为字符串，如 'float16', 'float32', 'float64',示例：**

- cast_to_floatx(x)将 numpy array 转换为默认的 Keras floatx 类型，x 为 numpy array，返回值也为 numpy array 但其数据类型变为 floatx。示例：
- >>> from keras import backend as K

```
>>> K.floatx()
'float32'
>>> arr = numpy.array([1.0, 2.0], dtype='float64')
>>> arr.dtype
dtype('float64')
>>> new_arr = K.cast_to_floatx(arr)
>>> new_arr
array([ 1.,  2.], dtype=float32)
>>> new_arr.dtype
dtype('float32')
```

- image_data_format() 返回默认的图像的维度顺序（‘channels_last’或‘channels_first’）
- set_image_data_format(data_format)设置图像的维度顺序（‘tf’或‘th’）,示例：

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

- is_keras_tensor()判断 x 是否是 keras tensor 对象的谓词函数>>> from keras import backend as K

```
>>> np_var = numpy.array([1, 2])
>>> K.is_keras_tensor(np_var)
False
>>> keras_var = K.variable(np_var)
>>> K.is_keras_tensor(keras_var)  # A variable is not a Tensor.
False
>>> keras_placeholder = K.placeholder(shape=(2, 4, 5))
>>> K.is_keras_tensor(keras_placeholder)  # A placeholder is a Tensor.
True
```

- get_uid(prefix='')获得默认计算图的 uid，依据给定的前缀提供一个唯一的 UID，参数为表示前缀的字符串，返回值为整数.
- reset_uids()
- clear_session()结束当前的 TF 计算图，并新建一个。有效的避免模型/层的混乱
- manual_variable_initialization(value) 指出变量应该以其默认值被初始化还是由用户手动初始化，参数 value 为布尔值，默认 False 代表变量由其默认值初始化
- (to be continued)
-
-

- Scikit-Learn 接口包装器

Scikit-Learn 接口包装器

- 我们可以通过包装器将 Sequential 模型（仅有一个输入）作为 Scikit-Learn 工作流的一部分，相关的包装器定义在 keras.wrappers.scikit_learn.py 中

目前，有两个包装器可用：

keras.wrappers.scikit_learn.KerasClassifier(build_fn=None, **sk_params)实现了 sklearn 的分类器接口

keras.wrappers.scikit_learn.KerasRegressor(build_fn=None, **sk_params)实现了 sklearn 的回归器接口

- **参数**
- build_fn：可调用的函数或类对象
- sk_params：模型参数和训练参数

build_fn 应构造、编译并返回一个 Keras 模型，该模型将稍后用于训练/测试。build_fn 的值可能为下列三种之一：

1. 一个函数
2. 一个具有 call 方法的类对象
3. None，代表你的类继承自 KerasClassifier 或 KerasRegressor，其 call 方法为其父类的 call 方法

sk_params 以模型参数和训练（超）参数作为参数。合法的模型参数为 build_fn 的参数。注意，‘build_fn’应提供其参数的默认值。所以我们不传递任何值给 sk_params 也可以创建一个分类器/回归器

sk_params 还接受用于调用 fit，predict，predict_proba 和 score 方法的参数，如 nb_epoch，batch_size 等。这些用于训练或预测的参数按如下顺序选择：

1. 传递给 fit，predict，predict_proba 和 score 的字典参数
2. 传递个 sk_params 的参数
3. keras.models.Sequential，fit，predict，predict_proba 和 score 的默认值

当使用 scikit-learn 的 grid_search 接口时，合法的可转换参数是你可以传递给 sk_params 的参数，包括训练参数。即，你可以使用 grid_search 来搜索最佳的 batch_size 或 nb_epoch 以及其他模型参数

- utils 工具
    - CustomObjectScope
    - keras.utils.generic_utils.CustomObjectScope()
    - 提供定制类的作用域，在该作用域内全局定制类能够被更改，但在作用域结束后将回到初始状态。 以 with 声明开头的代码将能够通过名字访问定制类的实例，在 with 的作用范围，这些定制类的变动将一直持续，在 with 作用域结束后，全局定制类的实例将回归其在 with 作用域前的状态。
    - ```
      with CustomObjectScope({"MyObject":MyObject}):
            layer = Dense(..., W_regularizer="MyObject")
            # save, load, etc. will recognize custom object by name
      ```
    - HDF5Matrix keras.utils.io_utils.HDF5Matrix(datapath, dataset, start=0, end=None, normalizer=None)
    - 这是一个使用 HDF5 数据集代替 Numpy 数组的方法

提供 start 和 end 参数可以进行切片，另外，还可以提供一个正规化函数或匿名函数，该函数将会在每片数据检索时自动调用。

- ```
  x_data = HDF5Matrix('input/file.hdf5', 'data')
  model.predict(x_data)
  ```
- datapath: 字符串，HDF5 文件的路径
- dataset: 字符串，在 datapath 路径下 HDF5 数据库名字
- start: 整数，想要的数据切片起点
- end: 整数，想要的数据切片终点
- normalizer: 在每个切片数据检索时自动调用的函数对象

- Sequence keras.utils.data_utils.Sequence()
- 序列数据的基类，例如一个数据集。 每个 Sequence 必须实现__getitem__和__len__方法

下面是一个例子：

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

```
__Here, `x_set` is list of path to the images__

# and `y_set` are the associated classes.

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 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)
```

- to_categorical `to_categorical(y, num_classes=None)`
- 将类别向量(从 0 到 nb_classes 的整数向量)映射为二值类别矩阵, 用于应用到以 `categorical_crossentropy` 为目标函数的模型中.
- normalize `normalize(x, axis=-1, order=2)`对 `numpy` 数组规范化，返回规范化后的数组 `order`：规范化方法，如 2 为 L2 范数
- convert_all_kernels_in_model
  `convert_all_kernels_in_model(model)`
- 将模型中全部卷积核在 Theano 和 TensorFlow 模式中切换
- **plot_model**
- `plot_model(model, to_file='model.png', show_shapes=False, show_layer_names=True)`
- 绘制模型的结构图
- **serialize_keras_object**
- `serialize_keras_object(instance)`
- 将 keras 对象序列化
- **deserialize_keras_object**
- `eserialize_keras_object(identifier, module_objects=None, custom_objects=None, printable_module_name='object')`
- 从序列中恢复 keras 对象
- **get_file**
- `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)`
- 从给定的 URL 中下载文件, 可以传递 MD5 值用于数据校验(下载后或已经缓存的数据均可)

默认情况下文件会被下载到`~/.keras` 中的 `cache_subdir` 文件夹，并将其文件名设为 `fname`，因此例如一个文件 `example.txt` 最终将会被存放在`~/.keras/datasets/example.txt~

tar,tar.gz.tar.bz 和 zip 格式的文件可以被提取，提供哈希码可以在下载后校验文件。命令喊程序 `shasum` 和 `sha256sum` 可以计算哈希值。

参数

- fname: 文件名，如果指定了绝对路径`/path/to/file.txt`,则文件将会保存到该位置。
- origin: 文件的 URL 地址
- untar: 布尔值,是否要进行解压
- md5_hash: MD5 哈希值,用于数据校验，支持 `sha256` 和 `md5` 哈希
- cache_subdir: 用于缓存数据的文件夹，若指定绝对路径`/path/to/folder` 则将存放在该路径下。
- hash_algorithm: 选择文件校验的哈希算法，可选项有'md5', 'sha256', 和'auto'. 默认'auto'自动检测使用的哈希算法
- extract: 若为 True 则试图提取文件，例如 tar 或 zip tries extracting the file as an Archive, like tar or zip.
- archive_format: 试图提取的文件格式，可选为'auto', 'tar', 'zip', 和 None. 'tar' 包括 tar, tar.gz, tar.bz 文件. 默认'auto'是['tar', 'zip']. None 或空列表将返回没有匹配。
- cache_dir: 缓存文件存放地在，参考 FAQ

返回值

下载后的文件地址

multi_gpu_model

```
keras.utils.multi_gpu_model(model, gpus)
```

将模型在多个 GPU 上复制

特别地，该函数用于单机多卡的数据并行支持，它按照下面的方式工作：

（1）将模型的输入分为多个子 batch （2）在每个设备上调用各自的模型，对各自的数据集运行 （3）将结果连接为一个大的 batch（在 CPU 上）

例如，你的 batch_size 是 64 而 gpus=2，则输入会被分为两个大小为 32 的子 batch，在两个 GPU 上分别运行，通过连接后返回大小为 64 的结果。 该函数线性的增加了训练速度，最高支持 8 卡并行。

该函数只能在 tf 后端下使用

参数如下：

- model: Keras 模型对象，为了避免 OOM 错误（内存不足），该模型应在 CPU 上构建，参考下面的例子。
- gpus: 大或等于 2 的整数，要并行的 GPU 数目。

该函数返回 Keras 模型对象，它看起来跟普通的 keras 模型一样，但实际上分布在多个 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

# Instantiate the base model
# (here, we do it on CPU, which is optional).
with tf.device('/cpu:0'):
    model = Xception(weights=None,
                     input_shape=(height, width, 3),
                     classes=num_classes)

# Replicates the model on 8 GPUs.
# This assumes that your machine has 8 available GPUs.
parallel_model = multi_gpu_model(model, gpus=8)
parallel_model.compile(loss='categorical_crossentropy',
                       optimizer='rmsprop')

# Generate dummy data.
x = np.random.random((num_samples, height, width, 3))
y = np.random.random((num_samples, num_classes))

# This `fit` call will be distributed on 8 GPUs.
# Since the batch size is 256, each GPU will process 32 samples.
parallel_model.fit(x, y, epochs=20, batch_size=256)
```

- For beginners
  - Keras FAQ：常见问题
  - 如何引用 Keras?
- 如何使 Keras 调用 GPU? 如果采用 TensorFlow 作为后端，当机器上有可用的 GPU 时，代码会自动调用 GPU 进行并行计算。如果使用 Theano 作为后端，可以通过以下方法设置：https://keras-cn.readthedocs.io/en/latest/for_beginners/FAQ/
- 如何在多张 GPU 卡上使用 Keras 我们建议有多张 GPU 卡可用时，使用 TnesorFlow 后端。

有两种方法可以在多张 GPU 上运行一个模型：数据并行/设备并行

大多数情况下，你需要的很可能是“数据并行”

- **数据并行**

数据并行将目标模型在多个设备上各复制一份，并使用每个设备上的复制品处理整个数据集的不同部分数据。Keras 在 `keras.utils.multi_gpu_model` 中提供有内置函数，该函数可以产生任意模型的数据并行版本，最高支持在 8 片 GPU 上并行。 请参考 utils 中的 multi_gpu_model 文档。 下面是一个例子：

- `from keras.utils import multi_gpu_model`

```
# Replicates `model` on 8 GPUs.
# This assumes that your machine has 8 available GPUs.
parallel_model = multi_gpu_model(model, gpus=8)
parallel_model.compile(loss='categorical_crossentropy',
                       optimizer='rmsprop')
```

```
# This `fit` call will be distributed on 8 GPUs.
# Since the batch size is 256, each GPU will process 32 samples.
parallel_model.fit(x, y, epochs=20, batch_size=256)
```

- **设备并行**

设备并行是在不同设备上运行同一个模型的不同部分，当模型含有多个并行结构，例如含有两个分支时，这种方式很适合。

这种并行方法可以通过使用 TensorFlow device scopes 实现，下面是一个例子：

```
# Model where a shared LSTM is used to encode two different sequences in
parallel
input_a = keras.Input(shape=(140, 256))
input_b = keras.Input(shape=(140, 256))

shared_lstm = keras.layers.LSTM(64)

# Process the first sequence on one GPU
with tf.device_scope('/gpu:0'):
    encoded_a = shared_lstm(tweet_a)
# Process the next sequence on another GPU
with tf.device_scope('/gpu:1'):
    encoded_b = shared_lstm(tweet_b)

# Concatenate results on CPU
with tf.device_scope('/cpu:0'):
    merged_vector = keras.layers.concatenate([encoded_a, encoded_b],
                                             axis=-1)
```

- "batch", "epoch"和"sample"都是啥意思？ Batch：中文为批，一个 batch 由若干条数据构成。batch 是进行网络优化的基本单位，网络参数的每一轮优化需要使用一个 batch。batch 中的样本是被并行处理的。与单个样本相比，一个 batch 的数据能更好的模拟数据集的分布，batch 越大则对输入数据分布模拟的越好，反应在网络训练上，则体现为能让网络训练的方向“更加正确”。但另一方面，一个 batch 也只能让网络的参数更新一次，因此网络参数的迭代会较慢。在测试网络的时候，应该在条件的允许的范围内尽量使用更大的 batch，这样计算效率会更高
- 设置 epoch 的主要作用是把模型的训练的整个训练过程分为若干个段，这样我们可以更好的观察和调整模型的训练。Keras 中，当指定了验证集时，每个 epoch 执行完后都会运行一次验证集以确定模型的性能。另外，我们可以使用回调函数在每个 epoch 的训练前后执行一些操作，如调整学习率，打印目前模型的一些信息等，详情请参考 Callback 一节。
- 如何保存 Keras 模型？ 我们不推荐使用 pickle 或 cPickle 来保存 Keras 模型

你可以使用`model.save(filepath)`将 Keras 模型和权重保存在一个 HDF5 文件中，该文件将包含：

- 模型的结构，以便重构该模型
- 模型的权重
- 训练配置（损失函数，优化器等）
- 优化器的状态，以便于从上次训练中断的地方开始

使用`keras.models.load_model(filepath)`来重新实例化你的模型，如果文件中存储了训练配置的话，该函数还会同时完成模型的编译

例子：

```
from keras.models import load_model

model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
del model  # deletes the existing model

# returns a compiled model
# identical to the previous one
model = load_model('my_model.h5')
```

如果你只是希望保存模型的结构，而不包含其权重或配置信息，可以使用：

```
# save as JSON
json_string = model.to_json()

# save as YAML
yaml_string = model.to_yaml()
```

这项操作将把模型序列化为 json 或 yaml 文件，这些文件对人而言也是友好的，如果需要的话你甚至可以手动打开这些文件并进行编辑。

当然，你也可以从保存好的 json 文件或 yaml 文件中载入模型：

```
# model reconstruction from JSON:
from keras.models import model_from_json
model = model_from_json(json_string)

# model reconstruction from YAML
model = model_from_yaml(yaml_string)
```

如果需要保存模型的权重，可通过下面的代码利用 HDF5 进行保存。注意，在使用前需要确保你已安装了 HDF5 和其 Python 库 h5py

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

如果你需要在代码中初始化一个完全相同的模型，请使用：

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

如果你需要加载权重到不同的网络结构（有些层一样）中，例如 fine-tune 或 transfer-learning，你可以通过层名字来加载模型：

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

例如：

```
"""
假如原模型为：
    model = Sequential()
    model.add(Dense(2, input_dim=3, name="dense_1"))
    model.add(Dense(3, name="dense_2"))
    ...
    model.save_weights(fname)
"""
# new model
model = Sequential()
model.add(Dense(2, input_dim=3, name="dense_1"))  # will be loaded
model.add(Dense(10, name="new_dense"))  # will not be loaded
```

```
# load weights from first model; will only affect the first layer, dense_1.
model.load_weights(fname, by_name=True)
```

-
- 为什么训练误差(loss)比测试误差高很多？ 一个 Keras 的模型有两个模式：训练模式和测试模式。一些正则机制，如 Dropout，L1/L2 正则项在测试模式下将不被启用。

另外，训练误差是训练数据每个 batch 的误差的平均。在训练过程中，每个 epoch 起始时的 batch 的误差要大一些，而后面的 batch 的误差要小一些。另一方面，每个 epoch 结束时计算的测试误差是由模型在 epoch 结束时的状态决定的，这时候的网络将产生较小的误差。

【Tips】可以通过定义回调函数将每个 epoch 的训练误差和测试误差并作图，如果训练误差曲线和测试误差曲线之间有很大的空隙，说明你的模型可能有过拟合的问题。当然，这个问题与 Keras 无关。

-
- 如何获取中间层的输出？ 一种简单的方法是创建一个新的 `Model`，使得它的输出是你想要的那个输出

```
from keras.models import Model

model = ...  # create the original model

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

此外，我们也可以建立一个 Keras 的函数来达到这一目的：

```
from keras import backend as K

# with a Sequential model
get_3rd_layer_output = K.function([model.layers[0].input],
                                  [model.layers[3].output])
layer_output = get_3rd_layer_output([X])[0]
```

当然，我们也可以直接编写 Theano 和 TensorFlow 的函数来完成这件事

注意，如果你的模型在训练和测试两种模式下不完全一致，例如你的模型中含有 Dropout 层，批规范化（BatchNormalization）层等组件，你需要在函数中传递一个 learning_phase 的标记，像这样：

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

# output in test mode = 0
layer_output = get_3rd_layer_output([X, 0])[0]

# output in train mode = 1
layer_output = get_3rd_layer_output([X, 1])[0]
```

-
- 如何利用 Keras 处理超过机器内存的数据集？ 可以使用 `model.train_on_batch(X,y)` 和 `model.test_on_batch(X,y)`。请参考模型

另外，也可以编写一个每次产生一个 batch 样本的生成器函数，并调用 `model.fit_generator(data_generator, samples_per_epoch, nb_epoch)`进行训练

这种方式在 Keras 代码包的 example 文件夹下 CIFAR10 例子里有示范，也可点击这里在 github 上浏览。

- Github code:

'''
#Train a simple deep CNN on the CIFAR10 small images dataset.
It gets to 75% validation accuracy in 25 epochs, and 79% after 50 epochs.
(it's still underfitting at that point, though).
'''

```
from __future__ import print_function
import keras
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
import os

batch_size = 32
num_classes = 10
epochs = 100
data_augmentation = True
num_predictions = 20
save_dir = os.path.join(os.getcwd(), 'saved_models')
model_name = 'keras_cifar10_trained_model.h5'

# The data, split between train and test sets:
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices.
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
```

```
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

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

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))

# initiate RMSprop optimizer
opt = keras.optimizers.RMSprop(learning_rate=0.0001, decay=1e-6)

# Let's train the model using RMSprop
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              validation_data=(x_test, y_test),
              shuffle=True)
else:
    print('Using real-time data augmentation.')
    # This will do preprocessing and realtime data augmentation:
    datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the dataset
        samplewise_std_normalization=False,  # divide each input by its std
```

```
        zca_whitening=False,  # apply ZCA whitening
        zca_epsilon=1e-06,  # epsilon for ZCA whitening
        rotation_range=0,  # randomly rotate images in the range (degrees, 0 to 180)
        # randomly shift images horizontally (fraction of total width)
        width_shift_range=0.1,
        # randomly shift images vertically (fraction of total height)
        height_shift_range=0.1,
        shear_range=0.,  # set range for random shear
        zoom_range=0.,  # set range for random zoom
        channel_shift_range=0.,  # set range for random channel shifts
        # set mode for filling points outside the input boundaries
        fill_mode='nearest',
        cval=0.,  # value used for fill_mode = "constant"
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False,  # randomly flip images
        # set rescaling factor (applied before any other transformation)
        rescale=None,
        # set function that will be applied on each input
        preprocessing_function=None,
        # image data format, either "channels_first" or "channels_last"
        data_format=None,
        # fraction of images reserved for validation (strictly between 0 and 1)
        validation_split=0.0)

    # Compute quantities required for feature-wise normalization
    # (std, mean, and principal components if ZCA whitening is applied).
    datagen.fit(x_train)

    # Fit the model on the batches generated by datagen.flow().
    model.fit_generator(datagen.flow(x_train, y_train,
                                     batch_size=batch_size),
                        epochs=epochs,
                        validation_data=(x_test, y_test),
                        workers=4)

# Save model and weights
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
model_path = os.path.join(save_dir, model_name)
model.save(model_path)
print('Saved trained model at %s ' % model_path)

# Score trained model.
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

- 当验证集的 loss 不再下降时，如何中断训练？ 可以定义 `EarlyStopping` 来提前终止训练

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

请参考回调函数

-
- 验证集是如何从训练集中分割出来的？ 如果在 `model.fit` 中设置 `validation_spilt` 的值，则可将数据分为训练集和验证集，例如，设置该值为 0.1，则训练集的最后 10%数据将作为验证集，设置其他数字同理。注意，原数据在进行验证集分割前并没有被 shuffle，所以这里的验证集严格的就是你输入数据最末的 x%。
- 训练数据在训练时会被随机洗乱吗？ 是的，如果 `model.fit` 的 `shuffle` 参数为真，训练的数据就会被随机洗乱。不设置时默认为真。训练数据会在每个 epoch 的训练中都重新洗乱一次。

验证集的数据不会被洗乱

-
- 如何在每个 epoch 后记录训练/测试的 loss 和正确率？ `model.fit` 在运行结束后返回一个 `History` 对象，其中含有的 `history` 属性包含了训练过程中损失函数的值以及其他度量指标。

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

- 如何使用状态 RNN（stateful RNN）？ 一个 RNN 是状态 RNN，意味着训练时每个 batch 的状态都会被重用于初始化下一个 batch 的初始状态。

当使用状态 RNN 时，有如下假设

- 所有的 batch 都具有相同数目的样本
- 如果 `X1` 和 `X2` 是两个相邻的 batch，那么对于任何 `i`，`X2[i]`都是 `X1[i]`的后续序列

要使用状态 RNN，我们需要

- 显式的指定每个 batch 的大小。可以通过模型的首层参数 `batch_input_shape` 来完成。`batch_input_shape` 是一个整数 tuple，例如(32,10,16)代表一个具有 10 个时间步，每步向量长为 16，每 32 个样本构成一个 batch 的输入数据格式。
- 在 RNN 层中，设置 `stateful=True`

要重置网络的状态，使用：

- `model.reset_states()`来重置网络中所有层的状态
- `layer.reset_states()`来重置指定层的状态

例子：

```
X  # this is our input data, of shape (32, 21, 16)
# we will feed it to our model in sequences of length 10

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')

# we train the network to predict the 11th timestep given the first 10:
model.train_on_batch(X[:, :10, :], np.reshape(X[:, 10, :], (32, 16)))

# the state of the network has changed. We can feed the follow-up sequences:
model.train_on_batch(X[:, 10:20, :], np.reshape(X[:, 20, :], (32, 16)))

# let's reset the states of the LSTM layer:
model.reset_states()

# another way to do it in this case:
model.layers[0].reset_states()
```

注意，`predict`，`fit`，`train_on_batch`，`predict_classes` 等方法都会更新模型中状态层的状态。这使得你不但可以进行状态网络的训练，也可以进行状态网络的预测。

- 如何“冻结”网络的层？ “冻结”一个层指的是该层将不参加网络训练，即该层的权重永不会更新。在进行 fine-tune 时我们经常会需要这项操作。 在使用固定的 embedding 层处理文本输入时，也需要这个技术。

可以通过向层的构造函数传递 `trainable` 参数来指定一个层是不是可训练的，如：

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

此外，也可以通过将层对象的 `trainable` 属性设为 `True` 或 `False` 来为已经搭建好的模型设置要冻结的层。 在设置完后，需要运行 `compile` 来使设置生效，例如：

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

frozen_model = Model(x, y)
# in the model below, the weights of `layer` will not be updated during training
frozen_model.compile(optimizer='rmsprop', loss='mse')

layer.trainable = True
trainable_model = Model(x, y)
# with this model the weights of the layer will be updated during training
# (which will also affect the above model since it uses the same layer instance)
trainable_model.compile(optimizer='rmsprop', loss='mse')

frozen_model.fit(data, labels)  # this does NOT update the weights of `layer`
trainable_model.fit(data, labels)  # this updates the weights of `layer`
```

- 如何从 Sequential 模型中去除一个层？ 可以通过调用`.pop()`来去除模型的最后一个层，反复调用 n 次即可去除模型后面的 n 个层

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

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

model.pop()
```

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

- 如何在 Keras 中使用预训练的模型 我们提供了下面这些图像分类的模型代码及预训练权重：
- VGG16
- VGG19
- ResNet50
- Inception v3

可通过 `keras.applications` 载入这些模型：

```
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

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

这些代码的使用示例请参考 `.Application` 模型的文档

使用这些预训练模型进行特征抽取或 fine-tune 的例子可以参考此博客

VGG 模型也是很多 Keras 例子的基础模型，如：

- Style-transfer
- Feature visualization
- Deep dream

- 如何在 Keras 中使用 HDF5 输入？ 你可以使用 keras.utils 中的 `HDF5Matrix` 类来读取 HDF5 输入，参考这里

可以直接使用 HDF5 数据库，示例

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

- Keras 的配置文件存储在哪里？
- 在使用 Keras 开发过程中，我如何获得可复现的结果？ 在开发模型中，有时取得可复现的结果是很有用的。例如，这可以帮助我们定位模型性能的改变是由模型本身引起的还是由于数据上的变化引起的。下面的代码展示了如何获得可复现的结果，该代码基于 Python3 的 tensorflow 后端

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

# The below is necessary in Python 3.2.3 onwards to
# have reproducible behavior for certain hash-based operations.
# See these references for further details:
# https://docs.python.org/3.4/using/cmdline.html#envvar-PYTHONHASHSEED
# https://github.com/fchollet/keras/issues/2280#issuecomment-306959926

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

# The below is necessary for starting Numpy generated random numbers
# in a well-defined initial state.
```

```
np.random.seed(42)

# The below is necessary for starting core Python generated random numbers
# in a well-defined state.

rn.seed(12345)

# Force TensorFlow to use single thread.
# Multiple threads are a potential source of
# non-reproducible results.
# For further details, see: https://stackoverflow.com/questions/42022950/which-
seeds-have-to-be-set-where-to-realize-100-reproducibility-of-training-res

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

from keras import backend as K

# The below tf.set_random_seed() will make random number generation
# in the TensorFlow backend have a well-defined initial state.
# For further details, see:
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)

# Rest of code follows ...
```

- 一些基本概念
    - 符号计算
    - 张量
    - data_format 这是一个无可奈何的问题，在如何表示一组彩色图片的问题上，Theano 和 TensorFlow 发生了分歧，'th'模式，也即 Theano 模式会把 100 张 RGB 三通道的 16×32（高为 16 宽为 32）彩色图表示为下面这种形式（100,3,16,32），Caffe 采取的也是这种方式。第 0 个维度是样本维，代表样本的数目，第 1 个维度是通道维，代表颜色通道数。后面两个就是高和宽了。这种 theano 风格的数据组织方法，称为“channels_first”，即通道维靠前。

而 TensorFlow，的表达形式是（100,16,32,3），即把通道维放在了最后，这种数据组织方式称为“channels_last”。

Keras 默认的数据组织形式在~/.keras/keras.json 中规定，可查看该文件的 `image_data_format` 一项查看，也可在代码中通过 K.image_data_format()函数返回，请在网络的训练和测试中保持维度顺序一致。

- 函数式模型 函数式模型算是本文档比较原创的词汇了，所以这里要说一下

在 Keras 0.x 中，模型其实有两种，一种叫 Sequential，称为序贯模型，也就是单输入单输出，一条路通到底，层与层之间只有相邻关系，跨层连接统统没有。这种模型编译速度快，操作上也比较简单。第二种模型称为 Graph，即图模型，这个模型支持多输入多输出，层与层之间想怎么连怎么连，但是编译速度慢。可以看到，Sequential 其实是 Graph 的一个特殊情况。

在 Keras1 和 Keras2 中，图模型被移除，而增加了了“functional model API”，这个东西，更加强调了 Sequential 是特殊情况这一点。一般的模型就称为 Model，然后如果你要用简单的 Sequential，OK，那还有一个快捷方式 Sequential。

由于 functional model API 在使用时利用的是“函数式编程”的风格，我们这里将其译为函数式模型。总而言之，只要这个东西接收一个或一些张量作为输入，然后输出的也是一个或一些张量，那不管它是什么鬼，统统都称作“模型”。

- batch 深度学习的优化算法，说白了就是梯度下降。每次的参数更新有两种方式。

第一种，遍历全部数据集算一次损失函数，然后算函数对各个参数的梯度，更新梯度。这种方法每更新一次参数都要把数据集里的所有样本都看一遍，计算量开销大，计算速度慢，不支持在线学习，这称为 Batch gradient descent，批梯度下降。

另一种，每看一个数据就算一下损失函数，然后求梯度更新参数，这个称为随机梯度下降，stochastic gradient descent。这个方法速度比较快，但是收敛性能不太好，可能在最优点附近晃来晃去，hit 不到最优点。两次参数的更新也有可能互相抵消掉，造成目标函数震荡的比较剧烈。

为了克服两种方法的缺点，现在一般采用的是一种折中手段，mini-batch gradient decent，小批的梯度下降，这种方法把数据分为若干个批，按批来更新参数，这样，一个批中的一组数据共同决定了本次梯度的方向，下降起来就不容易跑偏，减少了随机性。另一方面因为批的样本数与整个数据集相比小了很多，计算量也不是很大。

基本上现在的梯度下降都是基于 mini-batch 的，所以 Keras 的模块中经常会出现 batch_size，就是指这个。

顺便说一句，Keras 中用的优化器 SGD 是 stochastic gradient descent 的缩写，但不代表是一个样本就更新一回，还是基于 mini-batch 的。

- epochs
- 对新手友好的小说明

- 其他
- 一份简短的 Keras 介绍 Null
- Keras linux 关于计算机的硬件配置说明
- **2. Ubuntu 初始环境设置**

```
# 系统升级
>>> sudo apt update
>>> sudo apt upgrade
# 安装 python 基础开发包
>>> sudo apt install -y python-dev python-pip python-nose gcc g++ git gfortran
vim
```

安装运算加速库 打开终端输入：`>>> sudo apt install -y libopenblas-dev liblapack-dev libatlas-base-dev`

下载地址：https://developer.nvidia.com/cuda-downloads

```
>>> sudo dpkg -i cuda-repo-ubuntu1604-8-0-local-ga2_8.0.61-1_amd64.deb
>>> sudo apt update
>>> sudo apt -y install cuda 自动配置成功就好。
```

- 将 CUDA 路径添加至环境变量 在终端输入：`>>> sudo gedit /etc/profile`

- 在 profile 文件中添加：

```
export CUDA_HOME=/usr/local/cuda-8.0
export PATH=/usr/local/cuda-8.0/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64${LD_LIBRARY_PATH:+:$
{LD_LIBRARY_PATH}}
```

之后 source /etc/profile 即可

- 测试 在终端输入：

```
>>> nvcc -V
```

会得到相应的 nvcc 编译器相应的信息，那么 CUDA 配置成功了。(**记得重启系统**)

如果要进行 cuda 性能测试，可以进行：

```
>>> cd /usr/local/cuda/samples
>>> sudo make -j8
```

编译完成后，可以进 samples/bin/.../.../...的底层目录，运行各类实例。

4. 加速库 cuDNN（可选）

从官网下载需要注册账号申请，两三天批准。网盘搜索一般也能找到最新版。 Linux 目前最新的版本是 cudnn V6，但对于 tensorflow 的预编译版本还不支持这个最近版本，建议采用 5.1 版本，即是 cudnn-8.0-win-x64-v5.1-prod.zip。 下载解压出来是名为 cuda 的文件夹，里面有 bin、include、lib，将三个文件夹复制到安装 CUDA 的地方覆盖对应文件夹，在终端中输入：

```
>>> sudo cp include/cudnn.h /usr/local/cuda/include/
>>> sudo cp lib64/* /usr/local/cuda/lib64/
>>> cd /usr/local/cuda/lib64
>>> sudo ln -sf libcudnn.so.5.1.10 libcudnn.so.5
>>> sudo ln -sf libcudnn.so.5 libcudnn.so
>>> sudo ldconfig -v
```

Keras 框架搭建

相关开发包安装

在终端中输入:

```
>>> sudo pip install -U --pre pip setuptools wheel
>>> sudo pip install -U --pre numpy scipy matplotlib scikit-learn scikit-image
>>> sudo pip install -U --pre tensorflow-gpu
# >>> sudo pip install -U --pre tensorflow ## CPU版本
>>> sudo pip install -U --pre keras
```

安装完毕后，输入 python，然后输入：

```
>>> import tensorflow
>>> import keras
```

无错输出即可

Keras 中 mnist 数据集测试

下载 Keras 开发包

```
>>> git clone https://github.com/fchollet/keras.git
>>> cd keras/examples/
>>> python mnist_mlp.py
```

程序无错进行，至此，keras 安装完成。

- Keras windows not recommeded, but instruction is given. Pass.
- Keras 使用陷阱
    - TF 卷积核与 TH 卷积核 Keras 提供了两套后端，Theano 和 Tensorflow，这是一件幸福的事，就像手中拿着馒头，想蘸红糖蘸红糖，想蘸白糖蘸白糖

如果你从无到有搭建自己的一套网络，则大可放心。但如果你想使用一个已有网络，或把一个用 th/tf 训练的网络以另一种后端应用，在载入的时候你就应该特别小心了。

卷积核与所使用的后端不匹配，不会报任何错误，因为它们的 shape 是完全一致的，没有方法能够检测出这种错误。

在使用预训练模型时，一个建议是首先找一些测试样本，看看模型的表现是否与预计的一致。

如需对卷积核进行转换，可以使用 utils.convert_all_kernels_in_model 对模型的所有卷积核进行转换如果你不知道从哪里淘来一个预训练好的 BN 层，想把它的权重载入到 Keras 中，要小心参数的载入顺序。

一个典型的例子是，将 caffe 的 BN 层参数载入 Keras 中，caffe 的 BN 由两部分构成，bn 层的参数是 mean，std，scale 层的参数是 gamma，beta

按照 BN 的文章顺序，似乎载入 Keras BN 层的参数应该是[mean, std, gamma, beta]

然而不是的，Keras 的 BN 层参数顺序应该是[gamma, beta, mean, std]，这是因为 gamma 和 beta 是可训练的参数，而 mean 和 std 不是

Keras 的可训练参数在前，不可训练参数在后

错误的权重顺序不会引起任何报错，因为它们的 shape 完全相同

- 向 BN 层中载入权重 如果你不知道从哪里淘来一个预训练好的 BN 层，想把它的权重载入到 Keras 中，要小心参数的载入顺序。

一个典型的例子是，将 caffe 的 BN 层参数载入 Keras 中，caffe 的 BN 由两部分构成，bn 层的参数是 mean，std，scale 层的参数是 gamma，beta

按照 BN 的文章顺序，似乎载入 Keras BN 层的参数应该是[mean, std, gamma, beta]

然而不是的，Keras 的 BN 层参数顺序应该是[gamma, beta, mean, std]，这是因为 gamma 和 beta 是可训练的参数，而 mean 和 std 不是

Keras 的可训练参数在前，不可训练参数在后

错误的权重顺序不会引起任何报错，因为它们的 shape 完全相同

- shuffle 和 validation_split 的顺序 模型的 fit 函数有两个参数，shuffle 用于将数据打乱，validation_split 用于在没有提供验证集的时候，按一定比例从训练集中取出一部分作为验证集

这里有个陷阱是，程序是先执行 validation_split，再执行 shuffle 的，所以会出现这种情况：

假如你的训练集是有序的，比方说正样本在前负样本在后，又设置了 validation_split，那么你的验证集中很可能将全部是负样本

同样的，这个东西不会有任何错误报出来，因为 Keras 不可能知道你的数据有没有经过 shuffle，保险起见如果你的数据是没 shuffle 过的，最好手动 shuffle 一下

- Merge 层的层对象与函数方法 Keras 定义了一套用于融合张量的方法，位于 keras.layers.Merge，里面有两套工具，以大写字母开头的是 Keras Layer 类，使用这种工具是需要实例化一个 Layer 对象，然后再使用。以小写字母开头的是张量函数方法，本质上是对 Merge Layer 对象的一个包装，但使用更加方便一些。注意辨析。

- 未完待续

- Getting started
  - 快速开始函数式（Functional）模型 (below)
    - 第一个模型：全连接网络
    - 所有的模型都是可调用的，就像层一样
    - 多输入和多输出模型
    - 共享层
    - 层“节点”的概念
    - 更多的例子

快速开始函数式（Functional）模型

我们起初将 Functional 一词译作泛型，想要表达该类模型能够表达任意张量映射的含义，但表达的不是很精确，在 Keras 2 里我们将这个词改译为“函数式”，对函数式编程有所了解的同学应能够快速 get 到该类模型想要表达的含义。函数式模型称作 Functional，但它的类名是 Model，因此我们有时候也用 Model 来代表函数式模型。

Keras 函数式模型接口是用户定义多输出模型、非循环有向模型或具有共享层的模型等复杂模型的途径。一句话，只要你的模型不是类似 VGG 一样一条路走到黑的模型，或者你的模型需要多于一个的输出，那么你总应该选择函数式模型。函数式模型是最广泛的一类模型，序贯模型（Sequential）只是它的一种特殊情况。

这部分的文档假设你已经对 Sequential 模型已经比较熟悉

让我们从简单一点的模型开始

第一个模型：全连接网络

Sequential 当然是实现全连接网络的最好方式，但我们从简单的全连接网络开始，有助于我们学习这部分的内容。在开始前，有几个概念需要澄清：

- 层对象接受张量为参数，返回一个张量。
- 输入是张量，输出也是张量的一个框架就是一个模型，通过 Model 定义。
- 这样的模型可以被像 Keras 的 Sequential 一样被训练

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

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

所有的模型都是可调用的，就像层一样

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

```
x = Input(shape=(784,))
# This works, and returns the 10-way softmax we defined above.
y = model(x)
```

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

```
from keras.layers import TimeDistributed

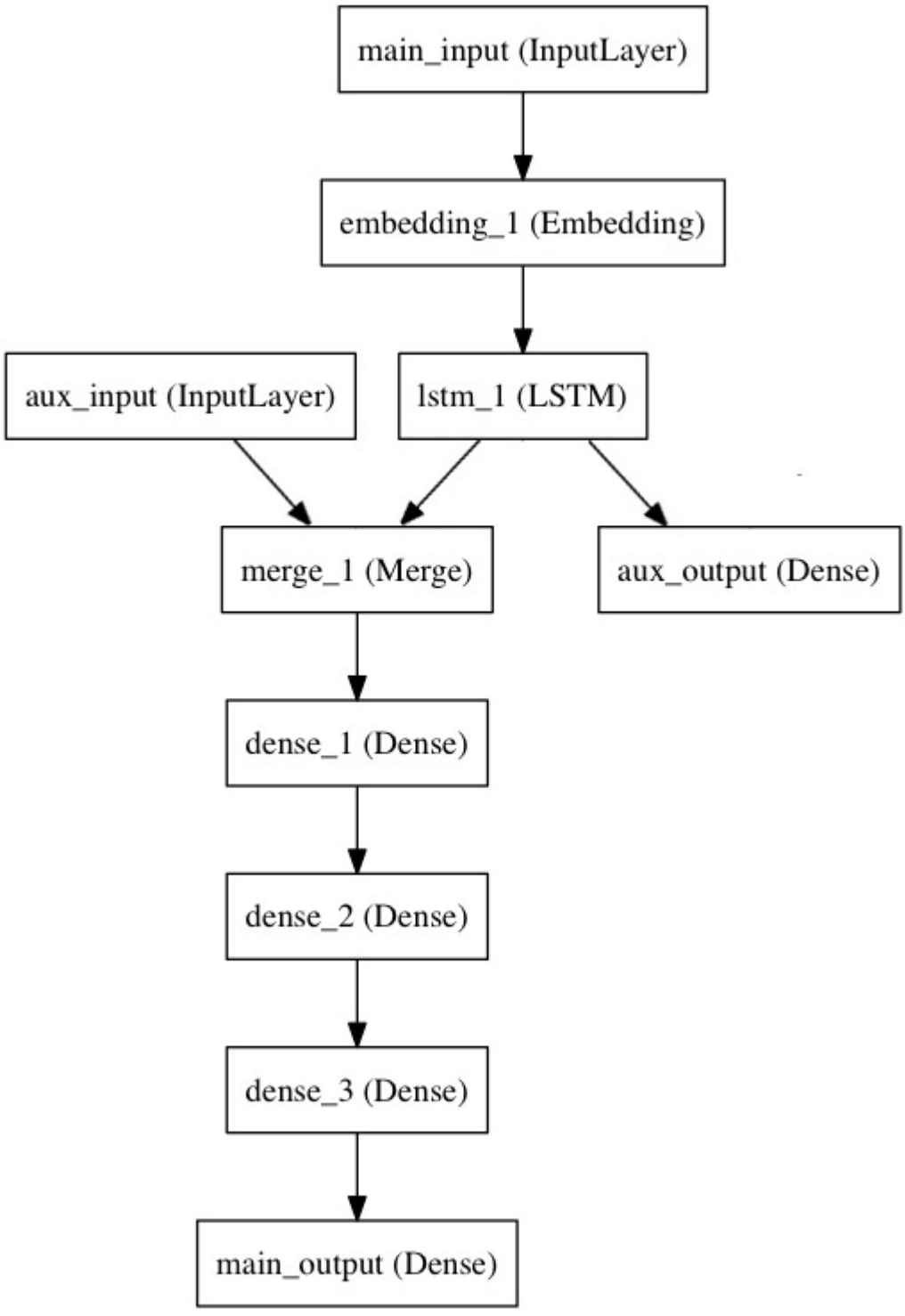
# Input tensor for sequences of 20 timesteps,
# each containing a 784-dimensional vector
input_sequences = Input(shape=(20, 784))

# This applies our previous model to every timestep in the input sequences.
# the output of the previous model was a 10-way softmax,
# so the output of the layer below will be a sequence of 20 vectors of size 10.
processed_sequences = TimeDistributed(model)(input_sequences)
```

多输入和多输出模型

使用函数式模型的一个典型场景是搭建多输入、多输出的模型。

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



让我们用函数式模型来实现这个框图

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

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

# Headline input: meant to receive sequences of 100 integers, between 1 and
10000.
# Note that we can name any layer by passing it a "name" argument.
main_input = Input(shape=(100,), dtype='int32', name='main_input')

# This embedding layer will encode the input sequence
# into a sequence of dense 512-dimensional vectors.
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)

# A LSTM will transform the vector sequence into a single vector,
# containing information about the entire sequence
lstm_out = LSTM(32)(x)
```

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

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

再然后，我们将 LSTM 与额外的输入数据串联起来组成输入，送入模型中：

```
auxiliary_input = Input(shape=(5,), name='aux_input')
x = keras.layers.concatenate([lstm_out, auxiliary_input])

# We stack a deep densely-connected network on top
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)

# And finally we add the main logistic regression layer
main_output = Dense(1, activation='sigmoid', name='main_output')(x)
```

最后，我们定义整个 2 输入，2 输出的模型：

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

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

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

编译完成后，我们通过传递训练数据和目标值训练该模型：

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

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

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

# And trained it via:
model.fit({'main_input': headline_data, 'aux_input': additional_data},
          {'main_output': labels, 'aux_output': labels},
          epochs=50, batch_size=32)
```

共享层

另一个使用函数式模型的场合是使用共享层的时候。

考虑微博数据，我们希望建立模型来判别两条微博是否是来自同一个用户，这个需求同样可以用来判断一个用户的两条微博的相似性。

一种实现方式是，我们建立一个模型，它分别将两条微博的数据映射到两个特征向量上，然后将特征向量串联并加一个 logistic 回归层，输出它们来自同一个用户的概率。这种模型的训练数据是一对对的微博。

因为这个问题是对称的，所以处理第一条微博的模型当然也能重用于处理第二条微博。所以这里我们使用一个共享的 LSTM 层来进行映射。

首先，我们将微博的数据转为（140，256）的矩阵，即每条微博有 140 个字符，每个单词的特征由一个 256 维的词向量表示，向量的每个元素为 1 表示某个字符出现，为 0 表示不出现，这是一个 one-hot 编码。

之所以是（140，256）是因为一条微博最多有 140 个字符，而扩展的 ASCII 码表编码了常见的 256 个字符。原文中此处为 Tweet，所以对外国人而言这是合理的。如果考虑中文字符，那一个单词的词向量就不止 256 了。

```
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))
```

若要对不同的输入共享同一层，就初始化该层一次，然后多次调用它

```
# This layer can take as input a matrix
# and will return a vector of size 64
shared_lstm = LSTM(64)

# When we reuse the same layer instance
# multiple times, the weights of the layer
# are also being reused
# (it is effectively *the same* layer)
encoded_a = shared_lstm(tweet_a)
encoded_b = shared_lstm(tweet_b)

# We can then concatenate the two vectors:
```

```
merged_vector = keras.layers.concatenate([encoded_a, encoded_b], axis=-1)

# And add a logistic regression on top
predictions = Dense(1, activation='sigmoid')(merged_vector)

# We define a trainable model linking the
# tweet inputs to the predictions
model = Model(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)
```

先暂停一下，看看共享层到底输出了什么，它的输出数据 shape 又是什么

层“节点”的概念

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

在上一版本的 Keras 中，你可以通过 `layer.get_output()`方法来获得层的输出张量，或者通过 `layer.output_shape` 获得其输出张量的 shape。这个版本的 Keras 你仍然可以这么做（除了 `layer.get_output()`被 `output` 替换）。但如果一个层与多个输入相连，会出现什么情况呢?

如果层只与一个输入相连，那没有任何困惑的地方。`.output` 将会返回该层唯一的输出

```
a = Input(shape=(140, 256))

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

assert lstm.output == encoded_a
```

但当层与多个输入相连时，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
```

上面这段代码会报错

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

通过下面这种调用方式即可解决

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

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

对于 input_shape 和 output_shape 也是一样，如果一个层只有一个节点，或所有的节点都有相同的输入或输出 shape，那么 input_shape 和 output_shape 都是没有歧义的，并也只返回一个值。但是，例如你把一个相同的 Conv2D 应用于一个大小为(32,32,3)的数据，然后又将其应用于一个(64,64,3)的数据，那么此时该层就具有了多个输入和输出的 shape，你就需要显式的指定节点的下标，来表明你想取的是哪个了

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

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

# Only one input so far, the following will work:
assert conv.input_shape == (None, 32, 32, 3)

conved_b = conv(b)
# now the `.input_shape` property wouldn't work, but this does:
assert conv.get_input_shape_at(0) == (None, 32, 32, 3)
assert conv.get_input_shape_at(1) == (None, 64, 64, 3)
```

更多的例子

代码示例依然是学习的最佳方式，这里是更多的例子

inception 模型

inception 的详细结构参见 Google 的这篇论文：Going Deeper with Convolutions

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

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

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

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

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

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

卷积层的残差连接

残差网络（Residual Network）的详细信息请参考这篇文章：Deep Residual Learning for Image Recognition

```
from keras.layers import Conv2D, Input

# input tensor for a 3-channel 256x256 image
x = Input(shape=(256, 256, 3))
```

```
# 3x3 conv with 3 output channels (same as input channels)
y = Conv2D(3, (3, 3), padding='same')(x)
# this returns x + y.
z = keras.layers.add([x, y])
```

共享视觉模型

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

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

# First, define the vision modules
digit_input = Input(shape=(27, 27, 1))
x = Conv2D(64, (3, 3))(digit_input)
x = Conv2D(64, (3, 3))(x)
x = MaxPooling2D((2, 2))(x)
out = Flatten()(x)

vision_model = Model(digit_input, out)

# Then define the tell-digits-apart model
digit_a = Input(shape=(27, 27, 1))
digit_b = Input(shape=(27, 27, 1))

# The vision model will be shared, weights and all
out_a = vision_model(digit_a)
out_b = vision_model(digit_b)

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

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

视觉问答模型

在针对一幅图片使用自然语言进行提问时，该模型能够提供关于该图片的一个单词的答案

这个模型将自然语言的问题和图片分别映射为特征向量，将二者合并后训练一个 logistic 回归层，从一系列可能的回答中挑选一个。

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

# First, let's define a vision model using a Sequential model.
# This model will encode an image into a vector.
vision_model = Sequential()
vision_model.add(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())
```

```
# Now let's get a tensor with the output of our vision model:
image_input = Input(shape=(224, 224, 3))
encoded_image = vision_model(image_input)

# Next, let's define a language model to encode the question into a vector.
# Each question will be at most 100 word long,
# and we will index words as integers from 1 to 9999.
question_input = Input(shape=(100,), dtype='int32')
embedded_question = Embedding(input_dim=10000, output_dim=256, input_length=100)
(question_input)
encoded_question = LSTM(256)(embedded_question)

# Let's concatenate the question vector and the image vector:
merged = keras.layers.concatenate([encoded_question, encoded_image])

# And let's train a logistic regression over 1000 words on top:
output = Dense(1000, activation='softmax')(merged)

# This is our final model:
vqa_model = Model(inputs=[image_input, question_input], outputs=output)

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

视频问答模型

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

```
from keras.layers import TimeDistributed

video_input = Input(shape=(100, 224, 224, 3))
# This is our video encoded via the previously trained vision_model (weights are
reused)
encoded_frame_sequence = TimeDistributed(vision_model)(video_input)  # the
output will be a sequence of vectors
encoded_video = LSTM(256)(encoded_frame_sequence)  # the output will be a vector

# This is a model-level representation of the question encoder, reusing the same
weights as before:
question_encoder = Model(inputs=question_input, outputs=encoded_question)

# Let's use it to encode the question:
video_question_input = Input(shape=(100,), dtype='int32')
encoded_video_question = question_encoder(video_question_input)

# And this is our video question answering model:
merged = 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)
```

快速开始序贯（Sequential）模型

序贯模型是多个网络层的线性堆叠，也就是“一条路走到黑”。

可以通过向 `Sequential` 模型传递一个 layer 的 list 来构造该模型：

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

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

也可以通过`.add()`方法一个个的将 layer 加入模型中：

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

指定输入数据的 shape

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

- 传递一个 `input_shape` 的关键字参数给第一层，`input_shape` 是一个 tuple 类型的数据，其中也可以填入 `None`，如果填入 `None` 则表示此位置可能是任何正整数。数据的 batch 大小不应包含在其中。
- 有些 2D 层，如 `Dense`，支持通过指定其输入维度 `input_dim` 来隐含的指定输入数据 shape, 是一个 Int 类型的数据。一些 3D 的时域层支持通过参数 `input_dim` 和 `input_length` 来指定输入 shape。
- 如果你需要为输入指定一个固定大小的 batch_size（常用于 stateful RNN 网络），可以传递 `batch_size` 参数到一个层中，例如你想指定输入张量的 batch 大小是 32，数据 shape 是（6，8），则你需要传递 `batch_size=32` 和 `input_shape=(6,8)`。

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

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

编译

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

- 优化器 optimizer：该参数可指定为已预定义的优化器名，如 `rmsprop`、`adagrad`，或一个 `Optimizer` 类的对象，详情见 optimizers
- 损失函数 loss：该参数为模型试图最小化的目标函数，它可为预定义的损失函数名，如 `categorical_crossentropy`、`mse`，也可以为一个损失函数。详情见 losses
- 指标列表 metrics：对分类问题，我们一般将该列表设置为 `metrics=['accuracy']`。指标可以是一个预定义指标的名字,也可以是一个用户定制的函数.指标函数应该返回单个张量,或一个完成 `metric_name - > metric_value` 映射的字典.请参考性能评估

```
# For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# For a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')

# For custom metrics
import keras.backend as K

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

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

训练

Keras 以 Numpy 数组作为输入数据和标签的数据类型。训练模型一般使用 `fit` 函数，该函数的详情见这里。下面是一些例子。

```
# For a single-input model with 2 classes (binary classification):

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'])

# Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
```

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

# Train the model, iterating on the data in batches of 32 samples
model.fit(data, labels, epochs=10, batch_size=32)

# For a single-input model with 10 classes (categorical classification):

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'])

# Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))

# Convert labels to categorical one-hot encoding
one_hot_labels = keras.utils.to_categorical(labels, num_classes=10)

# Train the model, iterating on the data in batches of 32 samples
model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

例子

这里是一些帮助你开始的例子

在 Keras 代码包的 examples 文件夹中，你将找到使用真实数据的示例模型：

- CIFAR10 小图片分类：使用 CNN 和实时数据提升
- IMDB 电影评论观点分类：使用 LSTM 处理成序列的词语
- Reuters（路透社）新闻主题分类：使用多层感知器（MLP）
- MNIST 手写数字识别：使用多层感知器和 CNN
- 字符级文本生成：使用 LSTM ...

基于多层感知器的 softmax 多分类：

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

# Generate dummy data
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) is a fully-connected layer with 64 hidden units.
# in the first layer, you must specify the expected input data shape:
# here, 20-dimensional vectors.
```

```
model.add(Dense(64, 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)
```

MLP 的二分类：

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

# Generate dummy data
x_train = np.random.random((1000, 20))
y_train = np.random.randint(2, size=(1000, 1))
x_test = np.random.random((100, 20))
y_test = np.random.randint(2, size=(100, 1))

model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

类似 VGG 的卷积神经网络：

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

# Generate dummy data
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()
```

```
# input: 100x100 images with 3 channels -> (100, 100, 3) tensors.
# this applies 32 convolution filters of size 3x3 each.
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)
```

使用 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)
```

使用 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',
```

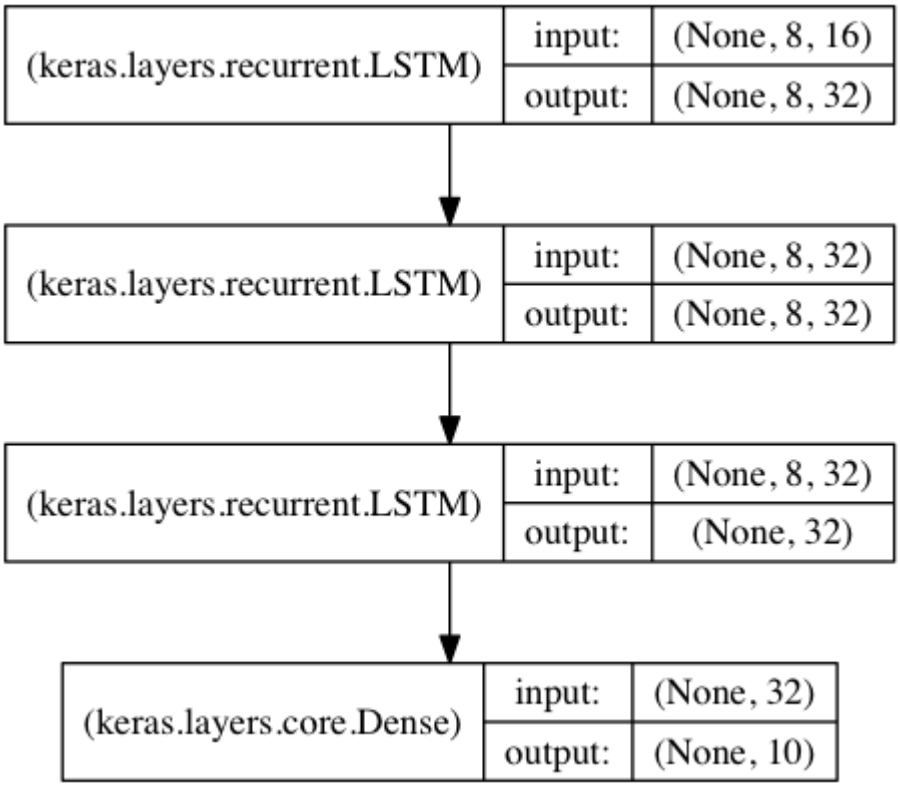
```
              optimizer='rmsprop',
              metrics=['accuracy'])

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

用于序列分类的栈式 LSTM

在该模型中，我们将三个 LSTM 堆叠在一起，是该模型能够学习更高层次的时域特征表示。

开始的两层 LSTM 返回其全部输出序列，而第三层 LSTM 只返回其输出序列的最后一步结果，从而其时域维度降低（即将输入序列转换为单个向量）




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

data_dim = 16
timesteps = 8
num_classes = 10

# expected input data shape: (batch_size, timesteps, data_dim)
model = Sequential()
model.add(LSTM(32, return_sequences=True,
               input_shape=(timesteps, data_dim)))  # returns a sequence of
vectors of dimension 32
model.add(LSTM(32, return_sequences=True))  # returns a sequence of vectors of
dimension 32
model.add(LSTM(32))  # return a single vector of dimension 32
model.add(Dense(10, activation='softmax'))

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

```
# Generate dummy training data
x_train = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, num_classes))

# Generate dummy validation data
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))
```

采用 stateful LSTM 的相同模型

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

请 FAQ 中关于 stateful LSTM 的部分获取更多信息

```
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

# Expected input batch shape: (batch_size, timesteps, data_dim)
# Note that we have to provide the full batch_input_shape since the network is
stateful.
# the sample of index i in batch k is the follow-up for the sample i in batch k-
1.
model = Sequential()
model.add(LSTM(32, return_sequences=True, stateful=True,
               batch_input_shape=(batch_size, timesteps, data_dim)))
model.add(LSTM(32, return_sequences=True, stateful=True))
model.add(LSTM(32, stateful=True))
model.add(Dense(10, activation='softmax'))

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

# Generate dummy training data
x_train = np.random.random((batch_size * 10, timesteps, data_dim))
y_train = np.random.random((batch_size * 10, num_classes))

# Generate dummy validation data
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))
```

- Layers
    - 关于 Keras 的“层”（Layer）

    - 所有的 Keras 层对象都有如下方法：
- `layer.get_weights()`：返回层的权重（numpy array）
- `layer.set_weights(weights)`：从 numpy array 中将权重加载到该层中，要求 numpy array 的形状与* `layer.get_weights()`的形状相同
- `layer.get_config()`：返回当前层配置信息的字典，层也可以借由配置信息重构:

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

或者：

```
from keras import layers

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

如果层仅有一个计算节点（即该层不是共享层），则可以通过下列方法获得输入张量、输出张量、输入数据的形状和输出数据的形状：

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

如果该层有多个计算节点（参考层计算节点和共享层）。可以使用下面的方法

- `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)`

    - 高级激活层 Advanced Activation
        - 高级激活层 Advanced Activation
            - LeakyReLU 层
              `keras.layers.advanced_activations.LeakyReLU(alpha=0.3)`

LeakyRelU 是修正线性单元（Rectified Linear Unit，ReLU）的特殊版本，当不激活时，LeakyReLU 仍然会有非零输出值，从而获得一个小梯度，避免 ReLU 可能出现的神经元“死亡”现象。即，`f(x)=alpha * x for x < 0, f(x) = x for x>=0`

参数

- alpha：大于 0 的浮点数，代表激活函数图像中第三象限线段的斜率

输入 shape

任意，当使用该层为模型首层时需指定 `input_shape` 参数

输出 shape

与输入相同

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

该层为参数化的 ReLU（Parametric ReLU），表达式是：`f(x) = alpha * x for x < 0`, `f(x) = x for x>=0`，此处的 `alpha` 为一个与 xshape 相同的可学习的参数向量。

参数

- alpha_initializer：alpha 的初始化函数
- alpha_regularizer：alpha 的正则项
- alpha_constraint：alpha 的约束项
- shared_axes：该参数指定的轴将共享同一组科学系参数，例如假如输入特征图是从 2D 卷积过来的，具有形如`(batch, height, width, channels)`这样的 shape，则或许你会希望在空域共享参数，这样每个 filter 就只有一组参数，设定 `shared_axes=[1,2]`可完成该目标

输入 shape

任意，当使用该层为模型首层时需指定 `input_shape` 参数

输出 shape

与输入相同

- ELU 层 ELU 层
  `keras.layers.advanced_activations.ELU(alpha=1.0)`

ELU 层是指数线性单元（Exponential Linera Unit），表达式为： 该层为参数化的 ReLU（Parametric ReLU），表达式是：`f(x) = alpha * (exp(x) - 1.) for x < 0`, `f(x) = x for x>=0`

参数

- alpha：控制负因子的参数

输入 shape

任意，当使用该层为模型首层时需指定 `input_shape` 参数

输出 shape

与输入相同

- ThresholdedReLU 层

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

该层是带有门限的 ReLU，表达式是：`f(x) = x for x > theta,f(x) = 0 otherwise`

参数

- theata：大或等于 0 的浮点数，激活门限位置

输入 shape

任意，当使用该层为模型首层时需指定 `input_shape` 参数

输出 shape

与输入相同

- 卷积层
  - Conv1D 层

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

一维卷积层（即时域卷积），用以在一维输入信号上进行邻域滤波。当使用该层作为首层时，需要提供关键字参数 `input_shape`。例如`(10,128)`代表一个长为 10 的序列，序列中每个信号为 128 向量。而`(None, 128)`代表变长的 128 维向量序列。

该层生成将输入信号与卷积核按照单一的空域（或时域）方向进行卷积。如果 `use_bias=True`，则还会加上一个偏置项，若 `activation` 不为 None，则输出为经过激活函数的输出。

参数

- filters：卷积核的数目（即输出的维度）
- kernel_size：整数或由单个整数构成的 list/tuple，卷积核的空域或时域窗长度
- strides：整数或由单个整数构成的 list/tuple，为卷积的步长。任何不为 1 的 strides 均与任何不为 1 的 dilation_rate 均不兼容

- padding：补 0 策略，为“valid”, “same” 或“causal”，“causal”将产生因果（膨胀的）卷积，即 output[t]不依赖于 input[t+1：]。当对不能违反时间顺序的时序信号建模时有用。参考 WaveNet: A Generative Model for Raw Audio, section 2.1.。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出 shape 与输入 shape 相同。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- dilation_rate：整数或由单个整数构成的 list/tuple，指定 dilated convolution 中的膨胀比例。任何不为 1 的 dilation_rate 均与任何不为 1 的 strides 均不兼容。
- use_bias:布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象

输入 shape

形如（samples，steps，input_dim）的 3D 张量

输出 shape

形如（samples，new_steps，nb_filter）的 3D 张量，因为有向量填充的原因，`steps` 的值会改变

【Tips】可以将 Convolution1D 看作 Convolution2D 的快捷版，对例子中（10，32）的信号进行 1D 卷积相当于对其进行卷积核为（filter_length, 32）的 2D 卷积。【@3rduncle】

- Conv2D 层

```
keras.layers.convolutional.Conv2D(filters,
kernel_size, strides=(1, 1), padding='valid',
data_format=None, dilation_rate=(1, 1),
activation=None, use_bias=True,
kernel_initializer='glorot_uniform',
bias_initializer='zeros',
```

```
                    kernel_regularizer=None, bias_regularizer=None,
                    activity_regularizer=None,
                    kernel_constraint=None, bias_constraint=None)
```

二维卷积层，即对图像的空域卷积。该层对二维输入进行滑动窗卷积，当使用该层作为第一层时，应提供`input_shape`参数。例如`input_shape = (128,128,3)`代表 128*128 的彩色 RGB 图像（`data_format='channels_last'`）

参数

- filters：卷积核的数目（即输出的维度）
- kernel_size：单个整数或由两个整数构成的 list/tuple，卷积核的宽度和长度。如为单个整数，则表示在各个空间维度的相同长度。
- strides：单个整数或由两个整数构成的 list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。任何不为 1 的 strides 均与任何不为 1 的 dilation_rate 均不兼容
- padding：补 0 策略，为“valid”, “same” 。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出 shape 与输入 shape 相同。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- dilation_rate：单个整数或由两个个整数构成的 list/tuple，指定 dilated convolution 中的膨胀比例。任何不为 1 的 dilation_rate 均与任何不为 1 的 strides 均不兼容。
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128x128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是`~/.keras/keras.json`中设置的值，若从未设置过，则为“channels_last”。
- use_bias:布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象

输入 shape

‘channels_first’模式下，输入形如（samples,channels，rows，cols）的 4D 张量

‘channels_last’模式下，输入形如（samples，rows，cols，channels）的 4D 张量

注意这里的输入 shape 指的是函数内部实现的输入 shape，而非函数接口应指定的 `input_shape`，请参考下面提供的例子。

输出 shape

‘channels_first’模式下，为形如（samples，nb_filter, new_rows, new_cols）的 4D 张量

‘channels_last’模式下，为形如（samples，new_rows, new_cols，nb_filter）的 4D 张量

输出的行列数可能会因为填充方法而改变

- SeparableConv2D 层

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

该层是在深度方向上的可分离卷积。

可分离卷积首先按深度方向进行卷积（对每个输入通道分别卷积），然后逐点进行卷积，将上一步的卷积结果混合到输出通道中。参数 `depth_multiplier` 控制了在 depthwise 卷积（第一步）的过程中，每个输入通道信号产生多少个输出通道。

直观来说，可分离卷积可以看做讲一个卷积核分解为两个小的卷积核，或看作 Inception 模块的一种极端情况。

当使用该层作为第一层时，应提供 `input_shape` 参数。例如 `input_shape = (3,128,128)` 代表 128*128 的彩色 RGB 图像

参数

- filters：卷积核的数目（即输出的维度）

- kernel_size：单个整数或由两个个整数构成的 list/tuple，卷积核的宽度和长度。如为单个整数，则表示在各个空间维度的相同长度。
- strides：单个整数或由两个整数构成的 list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。任何不为 1 的 strides 均与任何不为 1 的 dilation_rate 均不兼容
- padding：补 0 策略，为 “valid”, “same” 。 “valid”代表只进行有效的卷积，即对边界数据不处理。 “same”代表保留边界处的卷积结果，通常会导致输出 shape 与输入 shape 相同。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- dilation_rate：单个整数或由两个个整数构成的 list/tuple，指定 dilated convolution 中的膨胀比例。任何不为 1 的 dilation_rate 均与任何不为 1 的 strides 均不兼容。
- data_format：字符串， “channels_first”或 “channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering， “channels_last”对应原本的 “tf”， “channels_first”对应原本的 “th”。以 128x128 的 RGB 图像为例， “channels_first” 应将数据组织为（3,128,128），而 “channels_last”应将数据组织为（128,128,3）。该参数的默认值是`~/.keras/keras.json` 中设置的值，若从未设置过，则为 “channels_last”。
- use_bias:布尔值，是否使用偏置项
- depth_multiplier：在按深度卷积的步骤中，每个输入通道使用多少个输出通道
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- depthwise_regularizer：施加在按深度卷积的权重上的正则项，为 Regularizer 对象
- pointwise_regularizer：施加在按点卷积的权重上的正则项，为 Regularizer 对象
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象
- depthwise_constraint：施加在按深度卷积权重上的约束项，为 Constraints 对象
- pointwise_constraint 施加在按点卷积权重的约束项，为 Constraints 对象

输入 shape

‘channels_first’模式下，输入形如（samples,channels，rows，cols）的 4D 张量

‘channels_last’模式下，输入形如（samples，rows，cols，channels）的 4D 张量

注意这里的输入 shape 指的是函数内部实现的输入 shape，而非函数接口应指定的 `input_shape`，请参考下面提供的例子。

输出 shape

‘channels_first’模式下，为形如（samples，nb_filter, new_rows, new_cols）的 4D 张量

‘channels_last’模式下，为形如（samples，new_rows, new_cols，nb_filter）的 4D 张量

输出的行列数可能会因为填充方法而改变

- Conv2DTranspose 层

```
keras.layers.convolutional.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)
```

该层是转置的卷积操作（反卷积）。需要反卷积的情况通常发生在用户想要对一个普通卷积的结果做反方向的变换。例如，将具有该卷积层输出 shape 的 tensor 转换为具有该卷积层输入 shape 的 tensor。同时保留与卷积层兼容的连接模式。

当使用该层作为第一层时，应提供 `input_shape` 参数。例如 `input_shape = (3,128,128)` 代表 128*128 的彩色 RGB 图像

参数

- filters：卷积核的数目（即输出的维度）
- kernel_size：单个整数或由两个个整数构成的 list/tuple，卷积核的宽度和长度。如为单个整数，则表示在各个空间维度的相同长度。
- strides：单个整数或由两个整数构成的 list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。任何不为 1 的 strides 均与任何不为 1 的 dilation_rate 均不兼容
- padding：补 0 策略，为 “valid”, “same” 。 “valid”代表只进行有效的卷积，即对边界数据不处理。 “same”代表保留边界处的卷积结果，通常会导致输出 shape 与输入 shape 相同。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）

- dilation_rate：单个整数或由两个个整数构成的 list/tuple，指定 dilated convolution 中的膨胀比例。任何不为 1 的 dilation_rate 均与任何不为 1 的 strides 均不兼容。
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128x128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。
- use_bias:布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象

输入 shape

‘channels_first’模式下，输入形如（samples,channels，rows，cols）的 4D 张量

‘channels_last’模式下，输入形如（samples，rows，cols，channels）的 4D 张量

注意这里的输入 shape 指的是函数内部实现的输入 shape，而非函数接口应指定的 input_shape，请参考下面提供的例子。

输出 shape

‘channels_first’模式下，为形如（samples，nb_filter, new_rows, new_cols）的 4D 张量

‘channels_last’模式下，为形如（samples，new_rows, new_cols，nb_filter）的 4D 张量

输出的行列数可能会因为填充方法而改变

- Conv3D 层

```
keras.layers.convolutional.Conv3D(filters,
kernel_size, strides=(1, 1, 1),
padding='valid', data_format=None,
dilation_rate=(1, 1, 1), activation=None,
use_bias=True,
kernel_initializer='glorot_uniform',
```

```
                    bias_initializer='zeros',
                    kernel_regularizer=None, bias_regularizer=None,
                    activity_regularizer=None,
                    kernel_constraint=None, bias_constraint=None)
```

三维卷积对三维的输入进行滑动窗卷积，当使用该层作为第一层时，应提供 `input_shape` 参数。例如 `input_shape = (3,10,128,128)`代表对 10 帧 128*128 的彩色 RGB 图像进行卷积。数据的通道位置仍然有 `data_format` 参数指定。

参数

- filters：卷积核的数目（即输出的维度）
- kernel_size：单个整数或由 3 个整数构成的 list/tuple，卷积核的宽度和长度。如为单个整数，则表示在各个空间维度的相同长度。
- strides：单个整数或由 3 个整数构成的 list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。任何不为 1 的 strides 均与任何不为 1 的 dilation_rate 均不兼容
- padding：补 0 策略，为“valid”, “same” 。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出 shape 与输入 shape 相同。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- dilation_rate：单个整数或由 3 个个整数构成的 list/tuple，指定 dilated convolution 中的膨胀比例。任何不为 1 的 dilation_rate 均与任何不为 1 的 strides 均不兼容。
- data_format：字符串，“channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128x128x128 的数据为例，“channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是`~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。
- use_bias:布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象

- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象

输入 shape

‘channels_first’模式下，输入应为形如（samples，channels，input_dim1，input_dim2, input_dim3）的 5D 张量

‘channels_last’模式下，输入应为形如（samples，input_dim1，input_dim2, input_dim3，channels）的 5D 张量

这里的输入 shape 指的是函数内部实现的输入 shape，而非函数接口应指定的 `input_shape`。

- Cropping1D 层
  `keras.layers.convolutional.Cropping1D(cropping=(1, 1))`

在时间轴（axis1）上对 1D 输入（即时间序列）进行裁剪

参数

- cropping：长为 2 的 tuple，指定在序列的首尾要裁剪掉多少个元素

输入 shape

- 形如（samples，axis_to_crop，features）的 3D 张量

输出 shape

- 形如（samples，cropped_axis，features）的 3D 张量

- Cropping2D 层 Cropping2D 层

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

对 2D 输入（图像）进行裁剪，将在空域维度，即宽和高的方向上裁剪

参数

- cropping：长为 2 的整数 tuple，分别为宽和高方向上头部与尾部需要裁剪掉的元素数
- data_format：字符串， “channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering， “channels_last”对应原本的“tf”， “channels_first”对应原本的“th”。以 128x128 的 RGB 图像为例， “channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

输入 shape

形如（samples，depth, first_axis_to_crop, second_axis_to_crop）

输出 shape

形如(samples, depth, first_cropped_axis, second_cropped_axis)的 4D 张量

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

对 2D 输入（图像）进行裁剪

参数

- cropping：长为 3 的整数 tuple，分别为三个方向上头部与尾部需要裁剪掉的元素数
- data_format：字符串，“channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128x128x128 的数据为例，“channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是`~/.keras/keras.json`中设置的值，若从未设置过，则为“channels_last”。

输入 shape

形如 (samples, depth, first_axis_to_crop, second_axis_to_crop, third_axis_to_crop)的 5D 张量

输出 shape

形如(samples, depth, first_cropped_axis, second_cropped_axis, third_cropped_axis)的 5D 张

- UpSampling1D 层
  `keras.layers.convolutional.UpSampling1D(size=2)`

在时间轴上，将每个时间步重复`length`次

参数

- size：上采样因子

输入 shape

- 形如（samples，steps，features）的 3D 张量

输出 shape

- 形如（samples，upsampled_steps，features）的 3D 张量

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

将数据的行和列分别重复 size[0]和 size[1]次

参数

- size：整数 tuple，分别为行和列上采样因子
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128x128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是`~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。

输入 shape

‘channels_first’模式下，为形如（samples，channels, rows，cols）的 4D 张量

‘channels_last’模式下，为形如（samples，rows, cols，channels）的 4D 张量

输出 shape

‘channels_first’模式下，为形如（samples，channels, upsampled_rows, upsampled_cols）的 4D 张量

‘channels_last’模式下，为形如（samples，upsampled_rows, upsampled_cols，channels）的 4D 张量

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

将数据的三个维度上分别重复 size[0]、size[1]和 ize[2]次

本层目前只能在使用 Theano 为后端时可用

参数

- size：长为 3 的整数 tuple，代表在三个维度上的上采样因子
- data_format：字符串，“channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128x128x128 的数据为例，“channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是`~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。

输入 shape

‘channels_first’模式下，为形如（samples, channels, len_pool_dim1, len_pool_dim2, len_pool_dim3）的 5D 张量

‘channels_last’模式下，为形如（samples, len_pool_dim1, len_pool_dim2, len_pool_dim3，channels, ）的 5D 张量

输出 shape

‘channels_first’模式下，为形如（samples, channels, dim1, dim2, dim3）的 5D 张量

‘channels_last’模式下，为形如（samples, upsampled_dim1, upsampled_dim2, upsampled_dim3,channels,）的 5D 张量

- ZeroPadding1D 层
  `keras.layers.convolutional.ZeroPadding1D(padding=1)`

对 1D 输入的首尾端（如时域序列）填充 0，以控制卷积以后向量的长度

参数

- padding：整数，表示在要填充的轴的起始和结束处填充 0 的数目，这里要填充的轴是轴 1（第 1 维，第 0 维是样本数）

输入 shape

形如（samples，axis_to_pad，features）的 3D 张量

输出 shape

形如（samples，paded_axis，features）的 3D 张量

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

对 2D 输入（如图片）的边界填充 0，以控制卷积以后特征图的大小

参数

- padding：整数 tuple，表示在要填充的轴的起始和结束处填充 0 的数目，这里要填充的轴是轴 3 和轴 4（即在'th'模式下图像的行和列，在 ‘channels_last’模式下要填充的则是轴 2，3）
- data_format：字符串， “channels_first”或 “channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering， “channels_last”对应原本的 “tf”， “channels_first”对应原本的 “th”。以 128x128 的 RGB 图像为例， “channels_first”应将数据组织为（3,128,128），而 “channels_last”应将数据组织为（128,128,3）。该参数的默认值是`~/.keras/keras.json` 中设置的值，若从未设置过，则为 “channels_last”。

输入 shape

‘channels_first’模式下，形如（samples，channels，first_axis_to_pad，second_axis_to_pad）的 4D 张量

‘channels_last’模式下，形如（samples，first_axis_to_pad，second_axis_to_pad, channels）的 4D 张量

输出 shape

‘channels_first’模式下，形如（samples，channels，first_paded_axis，second_paded_axis）的 4D 张量

‘channels_last’模式下，形如（samples，first_paded_axis，second_paded_axis, channels）的 4D 张量

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

将数据的三个维度上填充 0

本层目前只能在使用 Theano 为后端时可用

参数

padding：整数 tuple，表示在要填充的轴的起始和结束处填充 0 的数目，这里要填充的轴是轴 3，轴 4 和轴 5， ‘channels_last’模式下则是轴 2，3 和 4

- data_format：字符串， “channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering， “channels_last”对应原本的“tf”， “channels_first”对应原本的“th”。以 128x128x128 的数据为例， “channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是`~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。

输入 shape

‘channels_first’模式下，为形如（samples, channels, first_axis_to_pad，first_axis_to_pad, first_axis_to_pad,）的 5D 张量

‘channels_last’模式下，为形如（samples, first_axis_to_pad，first_axis_to_pad, first_axis_to_pad, channels）的 5D 张量

输出 shape

‘channels_first’模式下，为形如（samples, channels, first_paded_axis，second_paded_axis, third_paded_axis,）的 5D 张量

‘channels_last’模式下，为形如（samples, len_pool_dim1, len_pool_dim2, len_pool_dim3，channels, ）的 5D 张量

- 常用层
  - Dense 层

- Activation 层
- Dropout 层
- Flatten 层
- Reshape 层
- Permute 层
- RepeatVector 层
- Lambda 层
- ActivityRegularizer 层

常用层对应于 core 模块，core 内部定义了一系列常用的网络层，包括全连接、激活层等

Dense 层

```
keras.layers.core.Dense(units, activation=None, use_bias=True,
kernel_initializer='glorot_uniform', bias_initializer='zeros',
kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
kernel_constraint=None, bias_constraint=None)
```

Dense 就是常用的全连接层，所实现的运算是 `output = activation(dot(input, kernel)+bias)`。其中 `activation` 是逐元素计算的激活函数，`kernel` 是本层的权值矩阵，`bias` 为偏置向量，只有当 `use_bias=True` 才会添加。

如果本层的输入数据的维度大于 2，则会先被压为与 `kernel` 相匹配的大小。

这里是一个使用示例：

```
# as first layer in a sequential model:
# as first layer in a sequential model:
model = Sequential()
model.add(Dense(32, input_shape=(16,)))
# now the model will take as input arrays of shape (*, 16)
# and output arrays of shape (*, 32)

# after the first layer, you don't need to specify
# the size of the input anymore:
model.add(Dense(32))
```

参数：

- units：大于 0 的整数，代表该层的输出维度。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- use_bias: 布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers

- bias_initializer：偏置向量初始化方法，为预定义初始化方法名的字符串，或用于初始化偏置向量的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象

输入

形如(batch_size, ..., input_dim)的 nD 张量，最常见的情况为(batch_size, input_dim)的 2D 张量

输出

形如(batch_size, ..., units)的 nD 张量，最常见的情况为(batch_size, units)的 2D 张量

Activation 层

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

激活层对一个层的输出施加激活函数

参数

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

输入 shape

任意，当使用激活层作为第一层时，要指定 `input_shape`

输出 shape

与输入 shape 相同

Dropout 层

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

为输入数据施加 Dropout。Dropout 将在训练过程中每次更新参数时按一定概率（rate）随机断开输入神经元，Dropout 层用于防止过拟合。

参数

- rate：0~1 的浮点数，控制需要断开的神经元的比例
- noise_shape：整数张量，为将要应用在输入上的二值 Dropout mask 的 shape，例如你的输入为 (batch_size, timesteps, features)，并且你希望在各个时间步上的 Dropout mask 都相同，则可传入 noise_shape=(batch_size, 1, features)。
- seed：整数，使用的随机数种子

参考文献

- Dropout: A Simple Way to Prevent Neural Networks from Overfitting

---

Flatten 层

`keras.layers.core.Flatten()`

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

例子

```
model = Sequential()
model.add(Convolution2D(64, 3, 3,
            border_mode='same',
            input_shape=(3, 32, 32)))
# now: model.output_shape == (None, 64, 32, 32)

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

Reshape 层

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

Reshape 层用来将输入 shape 转换为特定的 shape

参数

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

输入 shape

任意，但输入的 shape 必须固定。当使用该层为模型首层时，需要指定 `input_shape` 参数

输出 shape

`(batch_size,)+target_shape`

例子

```
# as first layer in a Sequential model
model = Sequential()
model.add(Reshape((3, 4), input_shape=(12,)))
# now: model.output_shape == (None, 3, 4)
# note: `None` is the batch dimension

# as intermediate layer in a Sequential model
model.add(Reshape((6, 2)))
# now: model.output_shape == (None, 6, 2)

# also supports shape inference using `-1` as dimension
model.add(Reshape((-1, 2, 2)))
# now: model.output_shape == (None, 3, 2, 2)
```

Permute 层

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

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

参数

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

例子

```
model = Sequential()
model.add(Permute((2, 1), input_shape=(10, 64)))
# now: model.output_shape == (None, 64, 10)
# note: `None` is the batch dimension
```

输入 shape

任意，当使用激活层作为第一层时，要指定 `input_shape`

输出 shape

与输入相同，但是其维度按照指定的模式重新排列

RepeatVector 层

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

RepeatVector 层将输入重复 n 次

参数

- n：整数，重复的次数

输入 shape

形如（nb_samples, features）的 2D 张量

输出 shape

形如（nb_samples, n, features）的 3D 张量

例子

```
model = Sequential()
model.add(Dense(32, input_dim=32))
# now: model.output_shape == (None, 32)
# note: `None` is the batch dimension

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

Lambda 层

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

本函数用以对上一层的输出施以任何 Theano/TensorFlow 表达式

参数

- function：要实现的函数，该函数仅接受一个变量，即上一层的输出
- output_shape：函数应该返回的值的 shape，可以是一个 tuple，也可以是一个根据输入 shape 计算输出 shape 的函数
- mask: 掩膜
- arguments：可选，字典，用来记录向函数中传递的其他关键字参数

例子

```
# add a x -> x^2 layer
model.add(Lambda(lambda x: x ** 2))
```

```
# add a layer that returns the concatenation
# of the positive part of the input and
# the opposite of the negative part

def antirectifier(x):
    x -= K.mean(x, axis=1, keepdims=True)
    x = K.l2_normalize(x, axis=1)
    pos = K.relu(x)
    neg = K.relu(-x)
    return K.concatenate([pos, neg], axis=1)

def antirectifier_output_shape(input_shape):
    shape = list(input_shape)
    assert len(shape) == 2  # only valid for 2D tensors
    shape[-1] *= 2
    return tuple(shape)

model.add(Lambda(antirectifier,
         output_shape=antirectifier_output_shape))
```

输入 shape

任意，当使用该层作为第一层时，要指定 `input_shape`

输出 shape

由 `output_shape` 参数指定的输出 shape，当使用 tensorflow 时可自动推断

ActivityRegularizer 层

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

经过本层的数据不会有任何变化，但会基于其激活值更新损失函数值

参数

- l1：1 范数正则因子（正浮点数）
- l2：2 范数正则因子（正浮点数）

输入 shape

任意，当使用该层作为第一层时，要指定 `input_shape`

输出 shape

与输入 shape 相同

Masking 层

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

使用给定的值对输入的序列信号进行“屏蔽”，用以定位需要跳过的时间步

对于输入张量的时间步，即输入张量的第 1 维度（维度从 0 开始算，见例子），如果输入张量在该时间步上都等于 `mask_value`，则该时间步将在模型接下来的所有层（只要支持 masking）被跳过（屏蔽）。

如果模型接下来的一些层不支持 masking，却接受到 masking 过的数据，则抛出异常。

例子

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

- 赋值 `x[:,3,:] = 0.，x[:,5,:] = 0.`
- 在 LSTM 层之前插入 `mask_value=0.`的 `Masking` 层

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

- Masking 层
- 嵌入层 EmbeddingEmbedding 层

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

嵌入层将正整数（下标）转换为具有固定大小的向量，如[[4],[20]]->[[0.25,0.1],[0.6,-0.2]]

Embedding 层只能作为模型的第一层

参数

- input_dim：大或等于 0 的整数，字典长度，即输入数据最大下标+1
- output_dim：大于 0 的整数，代表全连接嵌入的维度
- embeddings_initializer: 嵌入矩阵的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- embeddings_regularizer: 嵌入矩阵的正则项，为 Regularizer 对象
- embeddings_constraint: 嵌入矩阵的约束项，为 Constraints 对象
- mask_zero：布尔值，确定是否将输入中的‘0’看作是应该被忽略的‘填充’（padding）值，该参数在使用递归层处理变长输入时有用。设置为 `True` 的话，模型中后续的层必须都支持

masking，否则会抛出异常。如果该值为 True，则下标 0 在字典中不可用，input_dim 应设置为|vocabulary| + 1。

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

输入 shape

形如（samples，sequence_length）的 2D 张量

输出 shape

形如(samples, sequence_length, output_dim)的 3D 张量

例子

```
model = Sequential()
model.add(Embedding(1000, 64, input_length=10))
# the model will take as input an integer matrix of size (batch, input_length).
# the largest integer (i.e. word index) in the input should be no larger than
999 (vocabulary size).
# now model.output_shape == (None, 10, 64), where None is the batch dimension.

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)
```

- 局部连接层 LocallyConnceted LocallyConnected1D 层

```
keras.layers.local.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)
```

`LocallyConnected1D` 层与 `Conv1D` 工作方式类似，唯一的区别是不进行权值共享。即施加在不同输入位置的滤波器是不一样的。

参数

- filters：卷积核的数目（即输出的维度）
- kernel_size：整数或由单个整数构成的 list/tuple，卷积核的空域或时域窗长度
- strides：整数或由单个整数构成的 list/tuple，为卷积的步长。任何不为 1 的 strides 均与任何不为 1 的 dilation_rata 均不兼容
- padding：补 0 策略，目前仅支持 `valid`（大小写敏感），`same` 可能会在将来支持。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）

- dilation_rate：整数或由单个整数构成的 list/tuple，指定 dilated convolution 中的膨胀比例。任何不为 1 的 dilation_rata 均与任何不为 1 的 strides 均不兼容。
- use_bias:布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象

输入 shape

形如（samples，steps，input_dim）的 3D 张量

输出 shape

形如（samples，new_steps，nb_filter）的 3D 张量，因为有向量填充的原因，`steps` 的值会改变

LocallyConnected2D 层

```
keras.layers.local.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)
```

`LocallyConnected2D` 层与 `Convolution2D` 工作方式类似，唯一的区别是不进行权值共享。即施加在不同输入 patch 的滤波器是不一样的，当使用该层作为模型首层时，需要提供参数 `input_dim` 或 `input_shape` 参数。参数含义参考 `Convolution2D`。

参数

- filters：卷积核的数目（即输出的维度）
- kernel_size：单个整数或由两个整数构成的 list/tuple，卷积核的宽度和长度。如为单个整数，则表示在各个空间维度的相同长度。
- strides：单个整数或由两个整数构成的 list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。

- padding：补 0 策略，目前仅支持 `valid`（大小写敏感），`same` 可能会在将来支持。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128x128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是`~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。
- use_bias:布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象

输入 shape

‘channels_first’模式下，输入形如（samples,channels，rows，cols）的 4D 张量

‘channels_last’模式下，输入形如（samples，rows，cols，channels）的 4D 张量

注意这里的输入 shape 指的是函数内部实现的输入 shape，而非函数接口应指定的 `input_shape`，请参考下面提供的例子。

输出 shape

‘channels_first’模式下，为形如（samples，nb_filter, new_rows, new_cols）的 4D 张量

‘channels_last’模式下，为形如（samples，new_rows, new_cols，nb_filter）的 4D 张量

输出的行列数可能会因为填充方法而改变

例子

```
# apply a 3x3 unshared weights convolution with 64 output filters on a 32x32
image
# with `data_format="channels_last"`:
```

```
model = Sequential()
model.add(LocallyConnected2D(64, (3, 3), input_shape=(32, 32, 3)))
# now model.output_shape == (None, 30, 30, 64)
# notice that this layer will consume (30*30)*(3*3*3*64) + (30*30)*64 parameters

# add a 3x3 unshared weights convolution on top, with 32 output filters:
model.add(LocallyConnected2D(32, (3, 3)))
# now model.output_shape == (None, 28, 28, 32)
```

- Merge Merge 层提供了一系列用于融合两个层或两个张量的层对象和方法。以大写首字母开头的是 Layer 类，以小写字母开头的是张量的函数。小写字母开头的张量函数在内部实际上是调用了大写字母开头的层。

Add

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

添加输入列表的图层。

该层接收一个相同 shape 列表张量，并返回它们的和，shape 不变。

Example

```
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])  # equivalent to added =
keras.layers.add([x1, x2])

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

SubStract

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

两个输入的层相减。

它将大小至少为 2，相同 Shape 的列表张量作为输入，并返回一个张量（输入[0] - 输入[1]），也是相同的 Shape。

Example

```
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)
# Equivalent to subtracted = keras.layers.subtract([x1, x2])
subtracted = keras.layers.Subtract()([x1, x2])

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

Multiply

`keras.layers.Multiply()`

该层接收一个列表的同 shape 张量，并返回它们的逐元素积的张量，shape 不变。

Average

`keras.layers.Average()`

该层接收一个列表的同 shape 张量，并返回它们的逐元素均值，shape 不变。

Maximum

`keras.layers.Maximum()`

该层接收一个列表的同 shape 张量，并返回它们的逐元素最大值，shape 不变。

Concatenate

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

该层接收一个列表的同 shape 张量，并返回它们的按照给定轴相接构成的向量。

参数

- axis: 想接的轴
- **kwargs: 普通的 Layer 关键字参数

Dot

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

计算两个 tensor 中样本的张量乘积。例如，如果两个张量 a 和 b 的 shape 都为（batch_size, n），则输出为形如（batch_size,1）的张量，结果张量每个 batch 的数据都是 a[i,:]和 b[i,:]的矩阵（向量）点积。

参数

- axes: 整数或整数的 tuple，执行乘法的轴。
- normalize: 布尔值，是否沿执行成绩的轴做 L2 规范化，如果设为 True，那么乘积的输出是两个样本的余弦相似性。
- **kwargs: 普通的 Layer 关键字参数

add

`keras.layers.add(inputs)`

Add 层的函数式包装

参数：

- inputs: 长度至少为 2 的张量列表 A
- **kwargs: 普通的 Layer 关键字参数

返回值

输入列表张量之和

Example

```
import keras

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

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

subtract

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

Subtract 层的函数式包装

参数：

- inputs: 长度至少为 2 的张量列表 A
- **kwargs: 普通的 Layer 关键字参数

返回值

输入张量列表的差别

Example

```
import keras

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

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

multiply

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

Multiply 的函数式包装

参数：

- inputs: 长度至少为 2 的张量列表
- **kwargs: 普通的 Layer 关键字参数

返回值

输入列表张量之逐元素积

average

`keras.layers.average(inputs)`

Average 的函数包装

参数：

- inputs: 长度至少为 2 的张量列表
- **kwargs: 普通的 Layer 关键字参数

返回值

输入列表张量之逐元素均值

maximum

`keras.layers.maximum(inputs)`

Maximum 的函数包装

参数：

- inputs: 长度至少为 2 的张量列表
- **kwargs: 普通的 Layer 关键字参数

返回值

输入列表张量之逐元素均值

concatenate

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

Concatenate 的函数包装

参数

- inputs: 长度至少为 2 的张量列
- axis: 相接的轴
- **kwargs: 普通的 Layer 关键字参数

dot

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

Dot 的函数包装

参数

- inputs: 长度至少为 2 的张量列
- axes: 整数或整数的 tuple，执行乘法的轴。
- normalize: 布尔值，
- **kwargs: 普通的 Layer 关键字参数

    - 噪声层 Noise GaussianNoise 层

```
keras.layers.noise.GaussianNoise(stddev)
```

为数据施加 0 均值，标准差为 `stddev` 的加性高斯噪声。该层在克服过拟合时比较有用，你可以将它看作是随机的数据提升。高斯噪声是需要对输入数据进行破坏时的自然选择。

因为这是一个起正则化作用的层，该层只在训练时才有效。

参数

- stddev：浮点数，代表要产生的高斯噪声标准差

输入 shape

任意，当使用该层为模型首层时需指定 `input_shape` 参数

输出 shape

与输入相同

GaussianDropout 层

```
keras.layers.noise.GaussianDropout(rate)
```

为层的输入施加以 1 为均值，标准差为 `sqrt(rate/(1-rate)`的乘性高斯噪声

因为这是一个起正则化作用的层，该层只在训练时才有效。

参数

- rate：浮点数，断连概率，与 Dropout 层相同

输入 shape

任意，当使用该层为模型首层时需指定 `input_shape` 参数

输出 shape

与输入相同

参考文献

- Dropout: A Simple Way to Prevent Neural Networks from Overfitting

AlphaDropout

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

对输入施加 Alpha Dropout

Alpha Dropout 是一种保持输入均值和方差不变的 Dropout，该层的作用是即使在 dropout 时也保持数据的自规范性。 通过随机对负的饱和值进行激活，Alphe Drpout 与 selu 激活函数配合较好。

参数

- rate: 浮点数，类似 Dropout 的 Drop 比例。乘性 mask 的标准差将保证为 `sqrt(rate / (1 - rate))`.
- seed: 随机数种子

输入 shape

任意，当使用该层为模型首层时需指定 `input_shape` 参数

输出 shape

与输入相同

- （批）规范化 BatchNormalization BatchNormalization 层

```
keras.layers.normalization.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)
```

该层在每个 batch 上将前一层的激活值重新规范化，即使得其输出数据的均值接近 0，其标准差接近 1

参数

- axis: 整数，指定要规范化的轴，通常为特征轴。例如在进行 `data_format="channels_first` 的 2D 卷积后，一般会设 axis=1。
- momentum: 动态均值的动量
- epsilon：大于 0 的小浮点数，用于防止除 0 错误
- center: 若设为 True，将会将 beta 作为偏置加上去，否则忽略参数 beta
- scale: 若设为 True，则会乘以 gamma，否则不使用 gamma。当下一层是线性的时，可以设 False，因为 scaling 的操作将被下一层执行。
- beta_initializer：beta 权重的初始方法
- gamma_initializer: gamma 的初始化方法
- moving_mean_initializer: 动态均值的初始化方法
- moving_variance_initializer: 动态方差的初始化方法
- beta_regularizer: 可选的 beta 正则
- gamma_regularizer: 可选的 gamma 正则
- beta_constraint: 可选的 beta 约束
- gamma_constraint: 可选的 gamma 约束

输入 shape

任意，当使用本层为模型首层时，指定 `input_shape` 参数时有意义。

输出 shape

与输入 shape 相同

- 池化层 MaxPooling1D 层

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

对时域 1D 信号进行最大值池化

参数

- pool_size：整数，池化窗口大小
- strides：整数或 None，下采样因子，例如设 2 将会使得输出 shape 为输入的一半，若为 None 则默认值为 pool_size。
- padding： ‘valid’或者‘same’

输入 shape

- 形如（samples，steps，features）的 3D 张量

输出 shape

- 形如（samples，downsampled_steps，features）的 3D 张量

MaxPooling2D 层

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

为空域信号施加最大值池化

参数

- pool_size：整数或长为 2 的整数 tuple，代表在两个方向（竖直，水平）上的下采样因子，如取（2，2）将使图片在两个维度上均变为原长的一半。为整数意为各个维度值相同且为该数字。
- strides：整数或长为 2 的整数 tuple，或者 None，步长值。
- border_mode： ‘valid’或者 ‘same’
- data_format：字符串， “channels_first”或 “channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering， “channels_last”对应原本的 “tf”， “channels_first”对应原本的 “th”。以 128x128 的 RGB 图像为例， “channels_first”应将数据组织为（3,128,128），而 “channels_last”应将数据组织为（128,128,3）。该参数的默认值是`~/.keras/keras.json`中设置的值，若从未设置过，则为 “channels_last”。

输入 shape

‘channels_first’模式下，为形如（samples，channels, rows，cols）的 4D 张量

‘channels_last’模式下，为形如（samples，rows, cols，channels）的 4D 张量

输出 shape

‘channels_first’模式下，为形如（samples，channels, pooled_rows, pooled_cols）的 4D 张量

‘channels_last’模式下，为形如（samples，pooled_rows, pooled_cols，channels）的 4D 张量

MaxPooling3D 层

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

为 3D 信号（空域或时空域）施加最大值池化

本层目前只能在使用 Theano 为后端时可用

参数

- pool_size：整数或长为 3 的整数 tuple，代表在三个维度上的下采样因子，如取（2，2，2）将使信号在每个维度都变为原来的一半长。
- strides：整数或长为 3 的整数 tuple，或者 None，步长值。
- padding：‘valid’或者‘same’
- data_format：字符串，“channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128x128x128 的数据为例，“channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是`~/.keras/keras.json`中设置的值，若从未设置过，则为“channels_last”。

输入 shape

‘channels_first’模式下，为形如（samples, channels, len_pool_dim1, len_pool_dim2, len_pool_dim3）的 5D 张量

‘channels_last’模式下，为形如（samples, len_pool_dim1, len_pool_dim2, len_pool_dim3，channels, ）的 5D 张量

输出 shape

‘channels_first’模式下，为形如（samples, channels, pooled_dim1, pooled_dim2, pooled_dim3）的 5D 张量

‘channels_last’模式下，为形如（samples, pooled_dim1, pooled_dim2, pooled_dim3,channels,）的 5D 张量

AveragePooling1D 层

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

对时域 1D 信号进行平均值池化

参数

- pool_size：整数，池化窗口大小
- strides：整数或 None，下采样因子，例如设 2 将会使得输出 shape 为输入的一半，若为 None 则默认值为 pool_size。
- padding：‘valid’或者‘same’

输入 shape

- 形如（samples，steps，features）的 3D 张量

输出 shape

- 形如（samples，downsampled_steps，features）的 3D 张量

AveragePooling2D 层

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

为空域信号施加平均值池化

参数

- pool_size：整数或长为 2 的整数 tuple，代表在两个方向（竖直，水平）上的下采样因子，如取（2，2）将使图片在两个维度上均变为原长的一半。为整数意为各个维度值相同且为该数字。
- strides：整数或长为 2 的整数 tuple，或者 None，步长值。
- border_mode： ‘valid’或者 ‘same’
- data_format：字符串， “channels_first”或 “channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering， “channels_last”对应原本的 “tf”， “channels_first”对应原本的 “th”。以 128x128 的 RGB 图像为例， “channels_first”应将数据组织为（3,128,128），而 “channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为 “channels_last”。

输入 shape

‘channels_first’模式下，为形如（samples，channels, rows，cols）的 4D 张量

‘channels_last’模式下，为形如（samples，rows, cols，channels）的 4D 张量

输出 shape

‘channels_first’模式下，为形如（samples，channels, pooled_rows, pooled_cols）的 4D 张量

‘channels_last’模式下，为形如（samples，pooled_rows, pooled_cols，channels）的 4D 张量

AveragePooling3D 层

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

为 3D 信号（空域或时空域）施加平均值池化

本层目前只能在使用 Theano 为后端时可用

参数

- pool_size：整数或长为 3 的整数 tuple，代表在三个维度上的下采样因子，如取（2，2，2）将使信号在每个维度都变为原来的一半长。
- strides：整数或长为 3 的整数 tuple，或者 None，步长值。
- padding： ‘valid’或者 ‘same’
- data_format：字符串， “channels_first”或 “channels_last”之一，代表数据的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering， “channels_last”对应原本的 “tf”， “channels_first”对应原本的 “th”。以 128x128x128 的数据为例， “channels_first”应将数据组织为（3,128,128,128），而 “channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是`~/.keras/keras.json`中设置的值，若从未设置过，则为 “channels_last”。

‘channels_first’模式下，为形如（samples, channels, len_pool_dim1, len_pool_dim2, len_pool_dim3）的 5D 张量

‘channels_last’模式下，为形如（samples, len_pool_dim1, len_pool_dim2, len_pool_dim3，channels, ）的 5D 张量

输出 shape

‘channels_first’模式下，为形如（samples, channels, pooled_dim1, pooled_dim2, pooled_dim3）的 5D 张量

‘channels_last’模式下，为形如（samples, pooled_dim1, pooled_dim2, pooled_dim3,channels,）的 5D 张量

GlobalMaxPooling1D 层

`keras.layers.pooling.GlobalMaxPooling1D()`

对于时间信号的全局最大池化

输入 shape

- 形如（samples，steps，features）的 3D 张量

输出 shape

- 形如(samples, features)的 2D 张量

GlobalAveragePooling1D 层

`keras.layers.pooling.GlobalAveragePooling1D()`

为时域信号施加全局平均值池化

输入 shape

- 形如（samples，steps，features）的 3D 张量

输出 shape

- 形如(samples, features)的 2D 张量

GlobalMaxPooling2D 层

`keras.layers.pooling.GlobalMaxPooling2D(dim_ordering='default')`

为空域信号施加全局最大值池化

参数

- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128x128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

输入 shape

‘channels_first’模式下，为形如（samples，channels, rows，cols）的 4D 张量

‘channels_last’模式下，为形如（samples，rows, cols，channels）的 4D 张量

输出 shape

形如(nb_samples, channels)的 2D 张量

GlobalAveragePooling2D 层

`keras.layers.pooling.GlobalAveragePooling2D(dim_ordering='default')`

为空域信号施加全局平均值池化

参数

- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128x128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

输入 shape

‘channels_first’模式下，为形如（samples，channels, rows，cols）的 4D 张量

‘channels_last’模式下，为形如（samples，rows, cols，channels）的 4D 张量

输出 shape

形如(nb_samples, channels)的 2D 张量

- 循环层 Recurrent Recurrent 层

```
keras.layers.recurrent.Recurrent(return_sequences=False, go_backwards=False,
stateful=False, unroll=False, implementation=0)
```

这是循环层的抽象类，请不要在模型中直接应用该层（因为它是抽象类，无法实例化任何对象）。请使用它的子类 LSTM，GRU 或 SimpleRNN。

所有的循环层（LSTM,GRU,SimpleRNN）都继承本层，因此下面的参数可以在任何循环层中使用。

参数

- weights：numpy array 的 list，用以初始化权重。该 list 形如[(input_dim, output_dim),(output_dim, output_dim),(output_dim,)]
- return_sequences：布尔值，默认 False，控制返回类型。若为 True 则返回整个序列，否则仅返回输出序列的最后一个输出
- go_backwards：布尔值，默认为 False，若为 True，则逆向处理输入序列并返回逆序后的序列
- stateful：布尔值，默认为 False，若为 True，则一个 batch 中下标为 i 的样本的最终状态将会用作下一个 batch 同样下标的样本的初始状态。
- unroll：布尔值，默认为 False，若为 True，则循环层将被展开，否则就使用符号化的循环。当使用 TensorFlow 为后端时，循环网络本来就是展开的，因此该层不做任何事情。层展开会占用更多的内存，但会加速 RNN 的运算。层展开只适用于短序列。
- implementation：0，1 或 2， 若为 0，则 RNN 将以更少但是更大的矩阵乘法实现，因此在 CPU 上运行更快，但消耗更多的内存。如果设为 1，则 RNN 将以更多但更小的矩阵乘法实现，因此在 CPU 上运行更慢，在 GPU 上运行更快，并且消耗更少的内存。如果设为 2（仅 LSTM 和 GRU 可以设为 2），则 RNN 将把输入门、遗忘门和输出门合并为单个矩阵，以获得更加在

GPU 上更加高效的实现。注意，RNN dropout 必须在所有门上共享，并导致正则效果性能微弱降低。

- input_dim：输入维度，当使用该层为模型首层时，应指定该值（或等价的指定 input_shape)
- input_length：当输入序列的长度固定时，该参数为输入序列的长度。当需要在该层后连接 `Flatten` 层，然后又要连接 `Dense` 层时，需要指定该参数，否则全连接的输出无法计算出来。注意，如果循环层不是网络的第一层，你需要在网络的第一层中指定序列的长度（通过 `input_shape` 指定）。

输入 shape

形如（samples，timesteps，input_dim）的 3D 张量

输出 shape

如果 `return_sequences=True`：返回形如（samples，timesteps，output_dim）的 3D 张量

否则，返回形如（samples，output_dim）的 2D 张量

例子

```
# as the first layer in a Sequential model
model = Sequential()
model.add(LSTM(32, input_shape=(10, 64)))
# now model.output_shape == (None, 32)
# note: `None` is the batch dimension.

# the following is identical:
model = Sequential()
model.add(LSTM(32, input_dim=64, input_length=10))

# for subsequent layers, no need to specify the input size:
        model.add(LSTM(16))

# to stack recurrent layers, you must use return_sequences=True
# on any recurrent layer that feeds into another recurrent layer.
# note that you only need to specify the input size on the first layer.
model = Sequential()
model.add(LSTM(64, input_dim=64, input_length=10, return_sequences=True))
model.add(LSTM(32, return_sequences=True))
model.add(LSTM(10))
```

指定 RNN 初始状态的注意事项

可以通过设置 `initial_state` 用符号式的方式指定 RNN 层的初始状态。即，`initial_stat` 的值应该为一个 tensor 或一个 tensor 列表，代表 RNN 层的初始状态。

也可以通过设置 `reset_states` 参数用数值的方法设置 RNN 的初始状态，状态的值应该为 numpy 数组或 numpy 数组的列表，代表 RNN 层的初始状态。

屏蔽输入数据（Masking）

循环层支持通过时间步变量对输入数据进行 Masking，如果想将输入数据的一部分屏蔽掉，请使用 Embedding 层并将参数 `mask_zero` 设为 `True`。

使用状态 RNN 的注意事项

可以将 RNN 设置为 ‘stateful’，意味着由每个 batch 计算出的状态都会被重用于初始化下一个 batch 的初始状态。状态 RNN 假设连续的两个 batch 之中，相同下标的元素有一一映射关系。

要启用状态 RNN，请在实例化层对象时指定参数 `stateful=True`，并在 Sequential 模型使用固定大小的 batch：通过在模型的第一层传入 `batch_size=(...)`和 `input_shape` 来实现。在函数式模型中，对所有的输入都要指定相同的 `batch_size`。

如果要将循环层的状态重置，请调用`.reset_states()`，对模型调用将重置模型中所有状态 RNN 的状态。对单个层调用则只重置该层的状态。

SimpleRNN 层

```
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)
```

全连接 RNN 网络，RNN 的输出会被回馈到输入

参数

- units：输出维度
- activation：激活函数，为预定义的激活函数名（参考激活函数）
- use_bias: 布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象

- recurrent_regularizer：施加在循环核上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- recurrent_constraints：施加在循环核上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象
- dropout：0~1 之间的浮点数，控制输入线性变换的神经元断开比例
- recurrent_dropout：0~1 之间的浮点数，控制循环状态的线性变换的神经元断开比例
- 其他参数参考 Recurrent 的说明

参考文献

- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

---

GRU 层

```
keras.layers.recurrent.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)
```

门限循环单元（详见参考文献）

参数

- units：输出维度
- activation：激活函数，为预定义的激活函数名（参考激活函数）
- use_bias: 布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象

- recurrent_regularizer：施加在循环核上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- recurrent_constraints：施加在循环核上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象
- dropout：0~1 之间的浮点数，控制输入线性变换的神经元断开比例
- recurrent_dropout：0~1 之间的浮点数，控制循环状态的线性变换的神经元断开比例
- 其他参数参考 Recurrent 的说明

参考文献

- On the Properties of Neural Machine Translation: Encoder–Decoder Approaches
- Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling
- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

---

LSTM 层

```
keras.layers.recurrent.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)
```

Keras 长短期记忆模型，关于此算法的详情，请参考本教程

参数

- units：输出维度
- activation：激活函数，为预定义的激活函数名（参考激活函数）
- recurrent_activation: 为循环步施加的激活函数（参考激活函数）
- use_bias: 布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers

- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- recurrent_regularizer：施加在循环核上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- recurrent_constraints：施加在循环核上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象
- dropout：0~1 之间的浮点数，控制输入线性变换的神经元断开比例
- recurrent_dropout：0~1 之间的浮点数，控制循环状态的线性变换的神经元断开比例
- 其他参数参考 Recurrent 的说明

参考文献

- Long short-term memory（original 1997 paper）
- Learning to forget: Continual prediction with LSTM
- Supervised sequence labelling with recurrent neural networks
- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

ConvLSTM2D 层

```
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)
```

ConvLSTM2D 是一个 LSTM 网络，但它的输入变换和循环变换是通过卷积实现的

参数

- filters: 整数，输出的维度，该参数含义同普通卷积层的 filters
- kernel_size: 整数或含有 n 个整数的 tuple/list，指定卷积窗口的大小
- strides: 整数或含有 n 个整数的 tuple/list，指定卷积步长，当不等于 1 时，无法使用 dilation 功能，即 dialation_rate 必须为 1.
- padding: "valid" 或 "same" 之一

- data_format: * data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是 Keras 1.x 中的 image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以 128x128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是`~/.keras/keras.json`中设置的值，若从未设置过，则为“channels_last”。
- dilation_rate: 单个整数或由两个个整数构成的 list/tuple，指定 dilated convolution 中的膨胀比例。任何不为 1 的 dilation_rate 均与任何不为 1 的 strides 均不兼容。
- activation: activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- recurrent_activation: 用在 recurrent 部分的激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的 Theano 函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- use_bias: Boolean, whether the layer uses a bias vector.
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- recurrent_regularizer：施加在循环核上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- recurrent_constraints：施加在循环核上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象
- dropout：0~1 之间的浮点数，控制输入线性变换的神经元断开比例
- recurrent_dropout：0~1 之间的浮点数，控制循环状态的线性变换的神经元断开比例
- 其他参数参考 Recurrent 的说明

输入 shape

若 data_format='channels_first'， 为形如(samples,time, channels, rows, cols)的 5D tensor 若 data_format='channels_last' 为形如(samples,time, rows, cols, channels)的 5D tensor

输出 shape

if return_sequences： if data_format='channels_first'： 5D tensor (samples, time, filters, output_row, output_col) if data_format='channels_last'： 5D tensor (samples, time, output_row, output_col, filters) else if data_format ='channels_first' :4D tensor (samples, filters, output_row,

output_col) if data_format='channels_last' :4D tensor (samples, output_row, output_col, filters) (o_row 和 o_col 由 filter 和 padding 决定)

参考文献

Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting * 当前的实现不包含 cell 输出上的反馈循环（feedback loop）

SimpleRNNCell 层

```
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)
```

SinpleRNN 的 Cell 类

参数

- units：输出维度
- activation：激活函数，为预定义的激活函数名（参考激活函数）
- use_bias: 布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- recurrent_regularizer：施加在循环核上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- recurrent_constraints：施加在循环核上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象
- dropout：0~1 之间的浮点数，控制输入线性变换的神经元断开比例
- recurrent_dropout：0~1 之间的浮点数，控制循环状态的线性变换的神经元断开比例

GRUCell 层

```
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 的 Cell 类

参数

- units：输出维度
- activation：激活函数，为预定义的激活函数名（参考激活函数）
- use_bias: 布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- recurrent_regularizer：施加在循环核上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- recurrent_constraints：施加在循环核上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象
- dropout：0~1 之间的浮点数，控制输入线性变换的神经元断开比例
- recurrent_dropout：0~1 之间的浮点数，控制循环状态的线性变换的神经元断开比例
- 其他参数参考 Recurrent 的说明

LSTMCell 层

```
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 的 Cell 类

参数

- units：输出维度
- activation：激活函数，为预定义的激活函数名（参考激活函数）
- use_bias: 布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- recurrent_regularizer：施加在循环核上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- recurrent_constraints：施加在循环核上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象
- dropout：0~1 之间的浮点数，控制输入线性变换的神经元断开比例
- recurrent_dropout：0~1 之间的浮点数，控制循环状态的线性变换的神经元断开比例
- 其他参数参考 Recurrent 的说明

StackedRNNCells 层

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

这是一个 wrapper，用于将多个 recurrent cell 包装起来，使其行为类型单个 cell。该层用于实现搞笑的 stacked RNN

参数

- cells：list，其中每个元素都是一个 cell 对象

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.StackedRNNCells(cells)(inputs)
```

CuDNNGRU 层

```
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)
```

基于 CuDNN 的快速 GRU 实现，只能在 GPU 上运行，只能使用 tensoflow 为后端

参数

- units：输出维度
- activation：激活函数，为预定义的激活函数名（参考激活函数）
- use_bias: 布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- recurrent_regularizer：施加在循环核上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- recurrent_constraints：施加在循环核上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象
- dropout：0~1 之间的浮点数，控制输入线性变换的神经元断开比例

- recurrent_dropout：0~1 之间的浮点数，控制循环状态的线性变换的神经元断开比例
- 其他参数参考 Recurrent 的说明

CuDNNLSTM 层

```
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)
```

基于 CuDNN 的快速 LSTM 实现，只能在 GPU 上运行，只能使用 tensoflow 为后端

参数

- units：输出维度
- activation：激活函数，为预定义的激活函数名（参考激活函数）
- use_bias: 布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考 initializers
- kernel_regularizer：施加在权重上的正则项，为 Regularizer 对象
- bias_regularizer：施加在偏置向量上的正则项，为 Regularizer 对象
- recurrent_regularizer：施加在循环核上的正则项，为 Regularizer 对象
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象
- kernel_constraints：施加在权重上的约束项，为 Constraints 对象
- recurrent_constraints：施加在循环核上的约束项，为 Constraints 对象
- bias_constraints：施加在偏置上的约束项，为 Constraints 对象
- dropout：0~1 之间的浮点数，控制输入线性变换的神经元断开比例
- recurrent_dropout：0~1 之间的浮点数，控制循环状态的线性变换的神经元断开比例
- 其他参数参考 Recurrent 的说明
    - 包装器 Wrapper TimeDistributed 包装器

```
keras.layers.wrappers.TimeDistributed(layer)
```

该包装器可以把一个层应用到输入的每一个时间步上

参数

- layer：Keras 层对象

输入至少为 3D 张量，下标为 1 的维度将被认为是时间维

例如，考虑一个含有 32 个样本的 batch，每个样本都是 10 个向量组成的序列，每个向量长为 16，则其输入维度为`(32,10,16)`，其不包含 batch 大小的 `input_shape` 为`(10,16)`

我们可以使用包装器 `TimeDistributed` 包装 `Dense`，以产生针对各个时间步信号的独立全连接：

```
# as the first layer in a model
model = Sequential()
model.add(TimeDistributed(Dense(8), input_shape=(10, 16)))
# now model.output_shape == (None, 10, 8)

# subsequent layers: no need for input_shape
model.add(TimeDistributed(Dense(32)))
# now model.output_shape == (None, 10, 32)
```

程序的输出数据 shape 为`(32,10,8)`

使用 `TimeDistributed` 包装 `Dense` 严格等价于 `layers.TimeDistribuedDense`。不同的是包装器 `TimeDistribued` 还可以对别的层进行包装，如这里对 `Convolution2D` 包装：

```
model = Sequential()
model.add(TimeDistributed(Convolution2D(64, 3, 3), input_shape=(10, 3, 299,
299)))
```

Bidirectional 包装器

```
keras.layers.wrappers.Bidirectional(layer, merge_mode='concat', weights=None)
```

双向 RNN 包装器

参数

- layer：`Recurrent` 对象
- merge_mode：前向和后向 RNN 输出的结合方式，为 `sum`,`mul`,`concat`,`ave` 和 `None` 之一，若设为 None，则返回值不结合，而是以列表的形式返回

例子

```
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')
```

        - Writting layer 对于简单的定制操作，我们或许可以通过使用 `layers.core.Lambda` 层来完成。但对于任何具有可训练权重的定制层，你应该自己来实现。

这里是一个 Keras2 的层应该具有的框架结构(如果你的版本更旧请升级)，要定制自己的层，你需要实现下面三个方法

- `build(input_shape)`：这是定义权重的方法，可训练的权应该在这里被加入列表 `` `self.trainable_weights `` 中。其他的属性还包括 `self.non_trainabe_weights`（列表）和 `self.updates`（需要更新的形如（tensor, new_tensor）的 tuple 的列表）。你可以参考 `BatchNormalization` 层的实现来学习如何使用上面两个属性。这个方法必须设置 `self.built = True`，可通过调用 `super([layer],self).build()`实现
- `call(x)`：这是定义层功能的方法，除非你希望你写的层支持 masking，否则你只需要关心 `call` 的第一个参数：输入张量
- `compute_output_shape(input_shape)`：如果你的层修改了输入数据的 shape，你应该在这里指定 shape 变化的方法，这个函数使得 Keras 可以做自动 shape 推断

```
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)
```

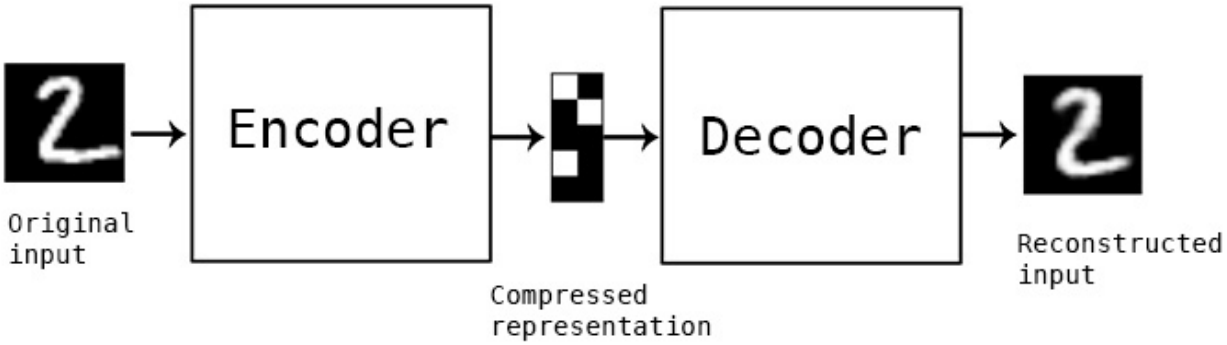
现存的 Keras 层代码可以为你的实现提供良好参考，阅读源代码吧!

- Legacy
    - 致谢 pass
    - Keras 后端 discussed
    - Keras:基于 Theano 和 TensorFlow 的深度学习库
    - No use
    - Scikit-Learn 接口包装器
    - Blog

- 自动编码器：各种各样的自动编码器

  本文地址：http://blog.keras.io/building-autoencoders-in-keras.html

本文作者：Francois Chollet



自动编码器是一种数据的压缩算法，其中数据的压缩和解压缩函数是 1）数据相关的,2）有损的，3）从样本中自动学习的。在大部分提到自动编码器的场合，压缩和解压缩的函数是通过神经网络实现的。

1）自动编码器是数据相关的（data-specific 或 data-dependent），这意味着自动编码器只能压缩那些与训练数据类似的数据。自编码器与一般的压缩算法，如 MPEG-2，MP3 等压缩算法不同，一般的通用算法只假设了数据是“图像”或“声音”，而没有指定是哪种图像或声音。比如，使用人脸训练出来的自动编码器在压缩别的图片，比如树木时性能很差，因为它学习到的特征是与人脸相关的。

2）自动编码器是有损的，意思是解压缩的输出与原来的输入相比是退化的，MP3，JPEG 等压缩算法也是如此。这与无损压缩算法不同。

3）自动编码器是从数据样本中自动学习的，这意味着很容易对指定类的输入训练出一种特定的编码器，而不需要完成任何新工作。

搭建一个自动编码器需要完成下面三样工作：搭建编码器，搭建解码器，设定一个损失函数，用以衡量由于压缩而损失掉的信息。编码器和解码器一般都是参数化的方程，并关于损失函数可导，典型情况是使用神经网络。编码器和解码器的参数可以通过最小化损失函数而优化，例如 SGD。

自编码器是一个好的数据压缩算法吗

通常情况下，使用自编码器做数据压缩，性能并不怎么样。以图片压缩为例，想要训练一个能和 JPEG 性能相提并论的自编码器非常困难，并且要达到这个性能，你还必须要把图片的类型限定在很小的一个范围内（例如 JPEG 不怎么行的某类图片）。自编码器依赖于数据的特性使得它在面对真实数据的压缩上并不可行，你只能在指定类型的数据上获得还可以的效果，但谁知道未来会有啥新需求？

那么，自编码器擅长做什么？

自编码器在实际应用中用的很少，2012 年人们发现在卷积神经网络中使用自编码器做逐层预训练可以训练深度网络，但很快人们发现良好的初始化策略在训练深度网络上要比费劲的逐层预训练有效得多，2014 年出现的 Batch Normalization 技术使得更深的网络也可以被有效训练，到了 2015 年底，通过使用残差学习（ResNet）我们基本上可以训练任意深度的神经网络。

目前自编码器的应用主要有两个方面，第一是数据去噪，第二是为进行可视化而降维。配合适当的维度和稀疏约束，自编码器可以学习到比 PCA 等技术更有意思的数据投影。

对于 2D 的数据可视化，t-SNE（读作 tee-snee）或许是目前最好的算法，但通常还是需要原数据的维度相对低一些。所以，可视化高维数据的一个好办法是首先使用自编码器将维度降低到较低的水平（如 32 维），然后再使用 t-SNE 将其投影在 2D 平面上。Keras 版本的 t-SNE 由 Kyle McDonald 实现了一下，放在了这里，另外 scikit-learn 也有一个简单实用的实现。

自编码器有什么卵用

自编码器的出名来自于网上很多机器学习课程的介绍，总而言之，一堆新手非常热爱自编码器而且怎么也玩不够，这就是这篇文章出现的意义【告诉你自编码器有什么卵用】。

自编码器吸引了一大批研究和关注的主要原因之一是很长时间一段以来它被认为是解决无监督学习的可能方案，即大家觉得自编码器可以在没有标签的时候学习到数据的有用表达。再说一次，自编码器并不是一个真正的无监督学习的算法，而是一个自监督的算法。自监督学习是监督学习的一个实例，其标签产生自输入数据。要获得一个自监督的模型，你需要想出一个靠谱的目标跟一个损失函数，问题来了，仅仅把目标设定为重构输入可能不是正确的选项。基本上，要求模型在像素级上精确重构输入不是机器学习的兴趣所在，学习到高级的抽象特征才是。事实上，当你的主要任务是分类、定位之类的任务时，那些对这类任务而言的最好的特征基本上都是重构输入时的最差的那种特征。

在应用自监督学习的视觉问题中，可能应用自编码器的领域有例如拼图，细节纹理匹配（从低分辨率的图像块中匹配其高分辨率的对应块）。下面这篇文章研究了拼图问题，其实很有意思，不妨一读。Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles.。此类问题的模型输入有些内置的假设，例如“视觉块比像素级的细节更重要”这样的，这种假设是普通的自编码器没有的。

Figure from Noroozi and Favaro (2016)

Fig. 1: What image representations do we learn by solving puzzles? Left: The image from which the tiles (marked with green lines) are extracted. Middle: A puzzle obtained by shuffling the tiles. Some tiles might be directly identifiable as object parts, but their identification is much more reliable once the correct ordering is found and the global figure emerges (Right).

使用 Keras 建立简单的自编码器

首先，先建立一个全连接的编码器和解码器

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

# this is the size of our encoded representations
```

```
encoding_dim = 32  # 32 floats -> compression of factor 24.5, assuming the input
is 784 floats

# this is our input placeholder
input_img = Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = Dense(784, activation='sigmoid')(encoded)

# this model maps an input to its reconstruction
autoencoder = Model(input=input_img, output=decoded)
```

当然我们可以单独的使用编码器和解码器：

```
# this model maps an input to its encoded representation
encoder = Model(input=input_img, output=encoded)

# create a placeholder for an encoded (32-dimensional) input
encoded_input = Input(shape=(encoding_dim,))
# retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# create the decoder model
decoder = Model(input=encoded_input, output=decoder_layer(encoded_input))
```

下面我们训练自编码器，来重构 MNIST 中的数字，这里使用逐像素的交叉熵作为损失函数，优化器为 adam

```
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

然后准备 MNIST 数据，将其归一化和向量化，然后训练：

```
from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print x_train.shape
print x_test.shape

autoencoder.fit(x_train, x_train,
                nb_epoch=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

50 个 epoch 后，看起来我们的自编码器优化的不错了，损失是 0.10，我们可视化一下重构出来的输出：

```
# encode and decode some digits
# note that we take them from the *test* set
# use Matplotlib (don't ask)
import matplotlib.pyplot as plt

encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)


n = 10  # how many digits we will display
```

```
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

这里是结果：

稀疏自编码器：为码字加上稀疏性约束

刚刚我们的隐层有 32 个神经元，这种情况下，一般而言自编码器学到的是 PCA 的一个近似（PCA 不想科普了）。但是如果我们对隐层单元施加稀疏性约束的话，会得到更为紧凑的表达，只有一小部分神经元会被激活。在 Keras 中，我们可以通过添加一个 activity_regularizer 达到对某层激活值进行约束的目的：

```
from keras import regularizers

encoding_dim = 32

input_img = Input(shape=(784,))
# add a Dense layer with a L1 activity regularizer
encoded = Dense(encoding_dim, activation='relu',
                activity_regularizer=regularizers.activity_l1(10e-5))(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)

autoencoder = Model(input=input_img, output=decoded)
```

因为我们添加了正则性约束，所以模型过拟合的风险降低，我们可以训练多几次，这次训练 100 个 epoch，得到损失为 0.11，多出来的 0.01 基本上是由于正则项造成的。可视化结果如下：

结果上没有毛线差别，区别在于编码出来的码字更加稀疏了。稀疏自编码器的在 10000 个测试图片上的码字均值为 3.33，而之前的为 7.30

深度自编码器：把自编码器叠起来

把多个自编码器叠起来，像这样：

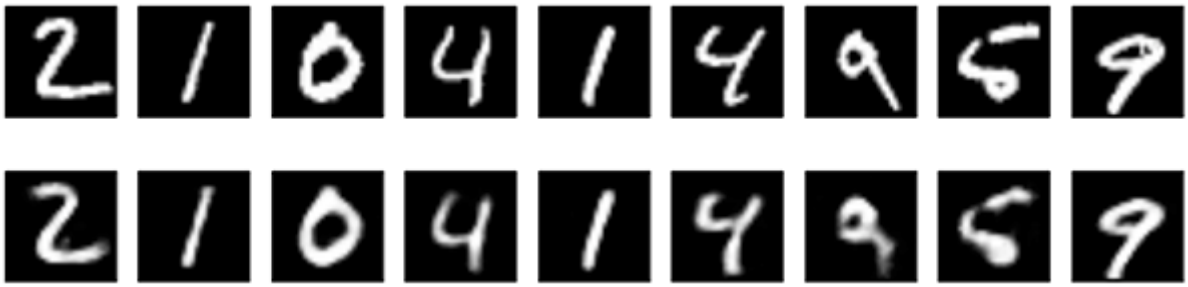
```
input_img = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(32, activation='relu')(encoded)

decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)

autoencoder = Model(input=input_img, output=decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
                nb_epoch=100,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

100 个 epoch 后，loss 大概是 0.097，比之前的模型好那么一丢丢



卷积自编码器：用卷积层搭建自编码器

当输入是图像时，使用卷积神经网络基本上总是有意义的。在现实中，用于处理图像的自动编码器几乎都是卷积自动编码器——又简单又快，棒棒哒

卷积自编码器的编码器部分由卷积层和 MaxPooling 层构成，MaxPooling 负责空域下采样。而解码器由卷积层和上采样层构成。

```
from keras.layers import Input, Dense, Convolution2D, MaxPooling2D, UpSampling2D
from keras.models import Model

input_img = Input(shape=(1, 28, 28))
```

```
x = Convolution2D(16, 3, 3, activation='relu', border_mode='same')(input_img)
x = MaxPooling2D((2, 2), border_mode='same')(x)
x = Convolution2D(8, 3, 3, activation='relu', border_mode='same')(x)
x = MaxPooling2D((2, 2), border_mode='same')(x)
x = Convolution2D(8, 3, 3, activation='relu', border_mode='same')(x)
encoded = MaxPooling2D((2, 2), border_mode='same')(x)

# at this point the representation is (8, 4, 4) i.e. 128-dimensional

x = Convolution2D(8, 3, 3, activation='relu', border_mode='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Convolution2D(8, 3, 3, activation='relu', border_mode='same')(x)
x = UpSampling2D((2, 2))(x)
x = Convolution2D(16, 3, 3, activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Convolution2D(1, 3, 3, activation='sigmoid', border_mode='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

我们使用 28*28*3 的原始 MNIST 图像（尽管看起来还是灰度图）训练网络，图片的像素被归一化到 0~1 之间。

```
from keras.datasets import mnist
import numpy as np

(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 1, 28, 28))
x_test = np.reshape(x_test, (len(x_test), 1, 28, 28))
```

为了可视化训练过程的损失情况，我们使用 TensorFlow 作为后端，这样就可以启用 TensorBoard 了。打开一个终端并启动 TensorBoard，TensorBoard 将读取位于/tmp/autoencoder 的日志文件：

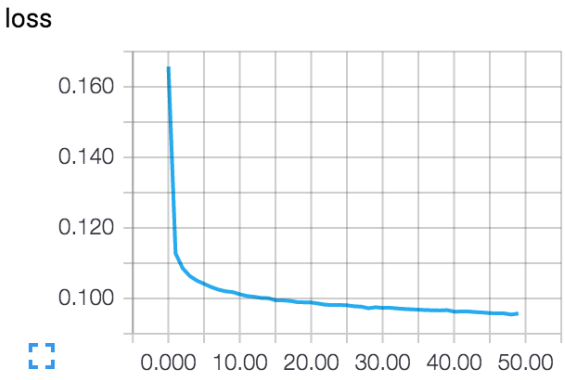
```
tensorboard --logdir=/tmp/autoencoder
```

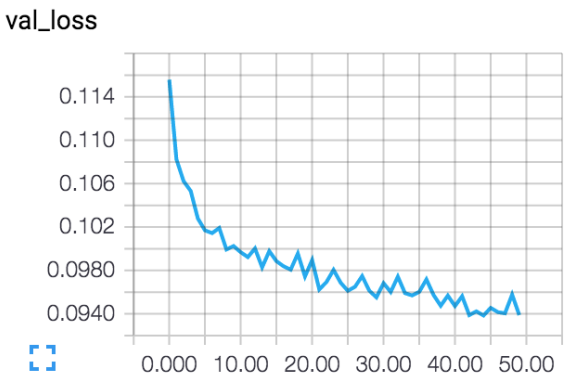
然后我们把模型训练 50 个 epoch，并在回调函数列表中传入 TensorBoard 回调函数，在每个 epoch 后回调函数将把训练的信息写入刚才的那个日志文件里，并被 TensorBoard 读取到

```
from keras.callbacks import TensorBoard

autoencoder.fit(x_train, x_train,
                nb_epoch=50,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test, x_test),
                callbacks=[TensorBoard(log_dir='/tmp/autoencoder')])
```

打开浏览器进入 http://0.0.0.0:6006 观测结果：



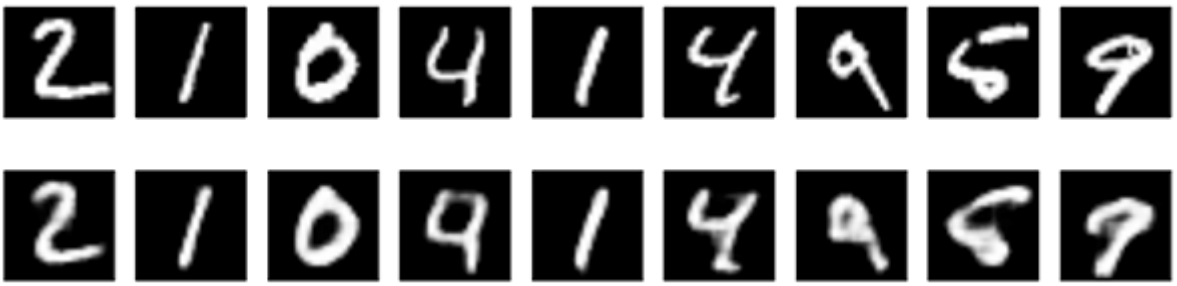


模型最后的 loss 是 0.094，要比之前的模型都要好得多，因为现在我们的编码器的表达表达能力更强了。

```
decoded_imgs = autoencoder.predict(x_test)

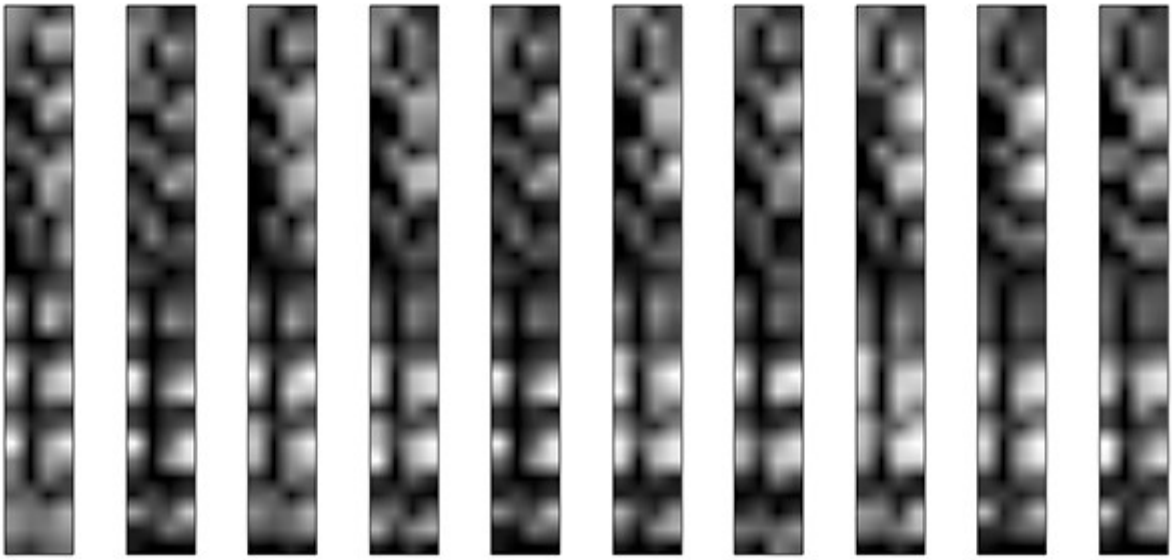
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



我们也可以看看中间的码字长什么样，这些码字的 shape 是 8*4*4，我们可以将其 reshape 成 4*32 看

```
n = 10
plt.figure(figsize=(20, 8))
for i in range(n):
    ax = plt.subplot(1, n, i)
    plt.imshow(encoded_imgs[i].reshape(4, 4 * 8).T)
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



使用自动编码器进行图像去噪

我们把训练样本用噪声污染，然后使解码器解码出干净的照片，以获得去噪自动编码器。首先我们把原图片加入高斯噪声，然后把像素值 clip 到 0~1

```
from keras.datasets import mnist
import numpy as np

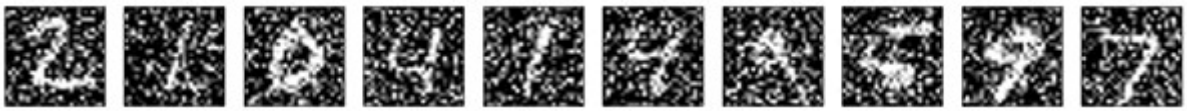
(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 1, 28, 28))
x_test = np.reshape(x_test, (len(x_test), 1, 28, 28))

noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=x_test.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

我们可以先看看被污染的照片长啥样：



- 和之前的卷积自动编码器相比，为了提高重构图质量，我们的模型稍有不同

```
input_img = Input(shape=(1, 28, 28))

x = Convolution2D(32, 3, 3, activation='relu', border_mode='same')(input_img)
x = MaxPooling2D((2, 2), border_mode='same')(x)
x = Convolution2D(32, 3, 3, activation='relu', border_mode='same')(x)
encoded = MaxPooling2D((2, 2), border_mode='same')(x)

# at this point the representation is (32, 7, 7)
```

```
x = Convolution2D(32, 3, 3, activation='relu', border_mode='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Convolution2D(32, 3, 3, activation='relu', border_mode='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Convolution2D(1, 3, 3, activation='sigmoid', border_mode='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

先来 100 个 epoch 的训练看看结果

```
autoencoder.fit(x_train_noisy, x_train,
                nb_epoch=100,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test_noisy, x_test),
                callbacks=[TensorBoard(log_dir='/tmp/tb', histogram_freq=0,
write_graph=False)])
```

结果如下，棒棒哒~

如果你将这个过程扩展到更大的卷积网络，你可以处理文档和声音的去噪，Kaggle 有一个或许你会感兴趣的数据集在这里

序列到序列的自动编码器

如果你的输入是序列而不是 2D 的图像，那么你可能想要使用针对序列的模型构造自编码器，如 LSTM。要构造基于 LSTM 的自编码器，首先我们需要一个 LSTM 的编码器来将输入序列变为一个向量，然后将这个向量重复 N 此，然后用 LSTM 的解码器将这个 N 步的时间序列变为目标序列。

这里我们不针对任何特定的数据库做这件事，只提供代码供读者参考

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

inputs = Input(shape=(timesteps, input_dim))
encoded = LSTM(latent_dim)(inputs)

decoded = RepeatVector(timesteps)(encoded)
decoded = LSTM(input, return_sequences=True)(decoded)

sequence_autoencoder = Model(inputs, decoded)
encoder = Model(inputs, encoded)
```

变分自编码器（Variational autoencoder，VAE）：编码数据的分布

编码自编码器是更现代和有趣的一种自动编码器，它为码字施加约束，使得编码器学习到输入数据的隐变量模型。隐变量模型是连接显变量集和隐变量集的统计模型，隐变量模型的假设是显变量是由隐变量的状态控制的，各个显变量之间条件独立。也就是说，变分编码器不再学习一个任意的函数，而是学习你的数据概率分布的一组参数。通过在这个概率分布中采样，你可以生成新的输入数据，即变分编码器是一个生成模型。

下面是变分编码器的工作原理：

首先，编码器网络将输入样本 x 转换为隐空间的两个参数，记作 z_mean 和 z_log_sigma。然后，我们随机从隐藏的正态分布中采样得到数据点 z，这个隐藏分布我们假设就是产生输入数据的那个分布。z = z_mean + exp(z_log_sigma)*epsilon，epsilon 是一个服从正态分布的张量。最后，使用解码器网络将隐空间映射到显空间，即将 z 转换回原来的输入数据空间。

参数藉由两个损失函数来训练，一个是重构损失函数，该函数要求解码出来的样本与输入的样本相似（与之前的自编码器相同），第二项损失函数是学习到的隐分布与先验分布的 KL 距离，作为一个正则。实际上把后面这项损失函数去掉也可以，尽管它对学习符合要求的隐空间和防止过拟合有帮助。

因为 VAE 是一个很复杂的例子，我们把 VAE 的代码放在了 github 上，在这里。在这里我们来一步步回顾一下这个模型是如何搭建的

首先，建立编码网络，将输入影射为隐分布的参数：

```
x = Input(batch_shape=(batch_size, original_dim))
h = Dense(intermediate_dim, activation='relu')(x)
z_mean = Dense(latent_dim)(h)
z_log_sigma = Dense(latent_dim)(h)
```

然后从这些参数确定的分布中采样，这个样本相当于之前的隐层值

```
def sampling(args):
    z_mean, z_log_sigma = args
    epsilon = K.random_normal(shape=(batch_size, latent_dim),
                              mean=0., std=epsilon_std)
    return z_mean + K.exp(z_log_sigma) * epsilon

# note that "output_shape" isn't necessary with the TensorFlow backend
# so you could write `Lambda(sampling)([z_mean, z_log_sigma])`
z = Lambda(sampling, output_shape=(latent_dim,))([z_mean, z_log_sigma])
```

最后，将采样得到的点映射回去重构原输入：

```
decoder_h = Dense(intermediate_dim, activation='relu')
decoder_mean = Dense(original_dim, activation='sigmoid')
h_decoded = decoder_h(z)
x_decoded_mean = decoder_mean(h_decoded)
```

到目前为止我们做的工作需要实例化三个模型：

- 一个端到端的自动编码器，用于完成输入信号的重构
- 一个用于将输入空间映射为隐空间的编码器

- 一个利用隐空间的分布产生的样本点生成对应的重构样本的生成器

```
# end-to-end autoencoder
vae = Model(x, x_decoded_mean)

# encoder, from inputs to latent space
encoder = Model(x, z_mean)

# generator, from latent space to reconstructed inputs
decoder_input = Input(shape=(latent_dim,))
_h_decoded = decoder_h(decoder_input)
_x_decoded_mean = decoder_mean(_h_decoded)
generator = Model(decoder_input, _x_decoded_mean)
```

我们使用端到端的模型训练，损失函数是一项重构误差，和一项 KL 距离

```
def vae_loss(x, x_decoded_mean):
    xent_loss = objectives.binary_crossentropy(x, x_decoded_mean)
    kl_loss = - 0.5 * K.mean(1 + z_log_sigma - K.square(z_mean) -
K.exp(z_log_sigma), axis=-1)
    return xent_loss + kl_loss

vae.compile(optimizer='rmsprop', loss=vae_loss)
```

现在使用 MNIST 库来训练变分编码器：

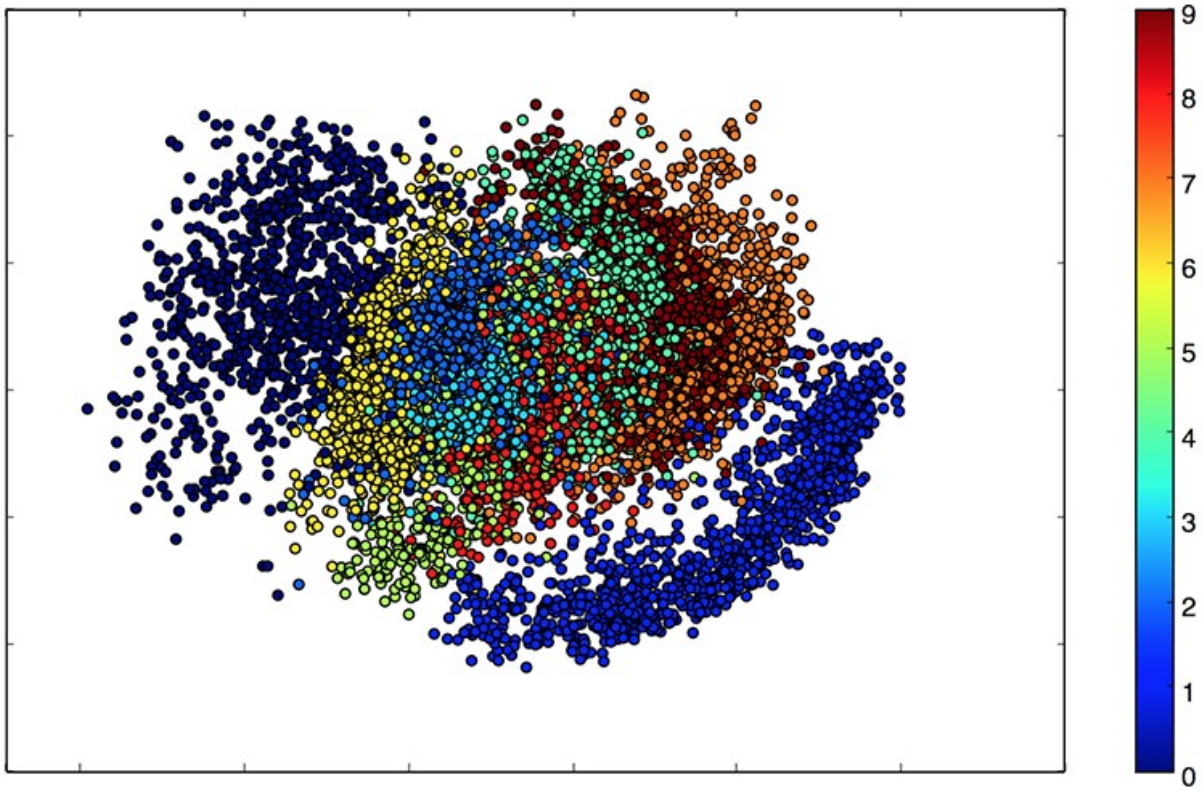
```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

vae.fit(x_train, x_train,
        shuffle=True,
        nb_epoch=nb_epoch,
        batch_size=batch_size,
        validation_data=(x_test, x_test))
```

因为我们的隐空间只有两维，所以我们可以可视化一下。我们来看看 2D 平面中不同类的近邻分布：

```
x_test_encoded = encoder.predict(x_test, batch_size=batch_size)
plt.figure(figsize=(6, 6))
plt.scatter(x_test_encoded[:, 0], x_test_encoded[:, 1], c=y_test)
plt.colorbar()
plt.show()
```



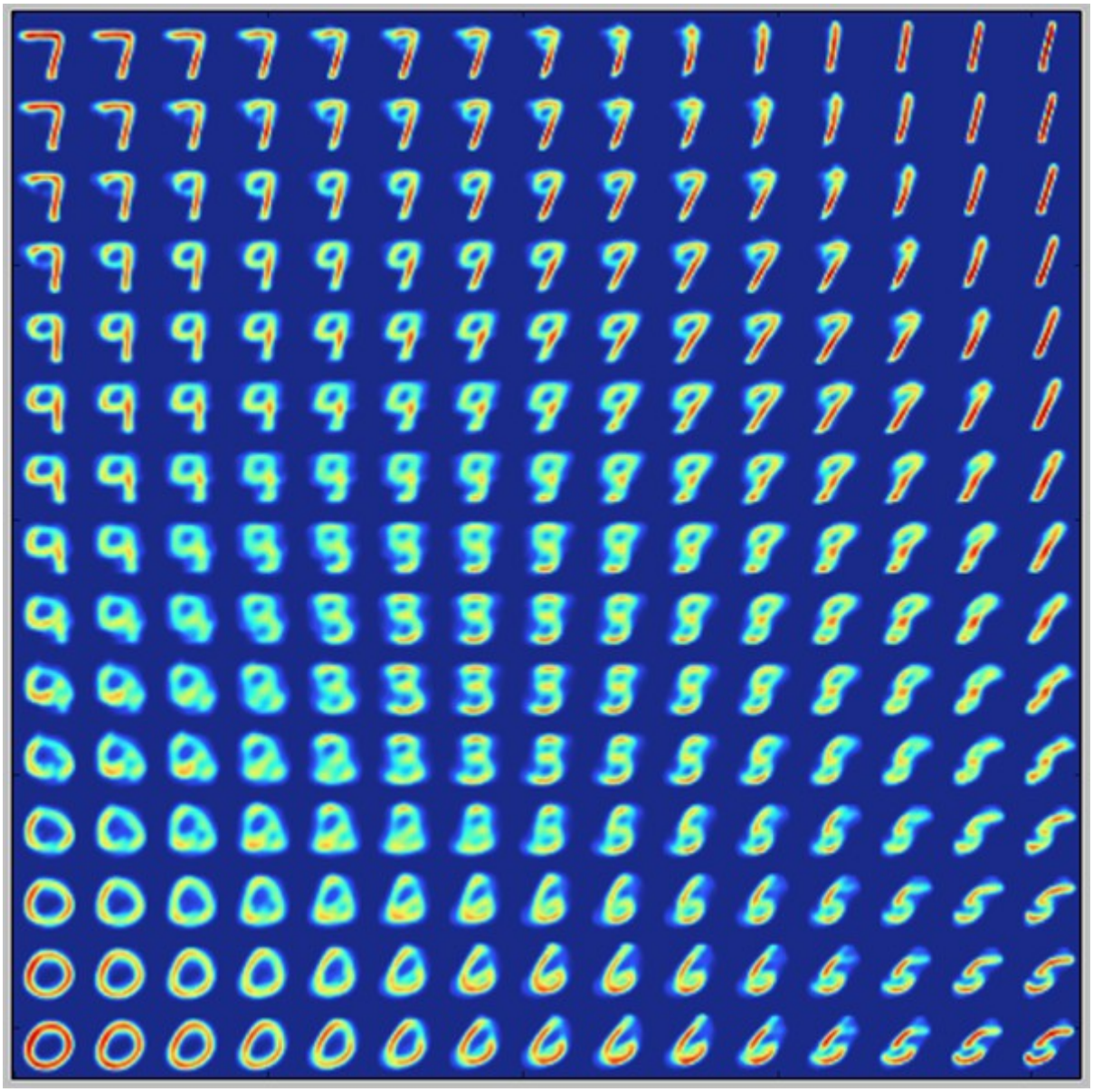
上图每种颜色代表一个数字，相近聚类的数字代表他们在结构上相似。

因为变分编码器是一个生成模型，我们可以用它来生成新数字。我们可以从隐平面上采样一些点，然后生成对应的显变量，即 MNIST 的数字：

```
# display a 2D manifold of the digits
n = 15  # figure with 15x15 digits
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
# we will sample n points within [-15, 15] standard deviations
grid_x = np.linspace(-15, 15, n)
grid_y = np.linspace(-15, 15, n)

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]]) * epsilon_std
        x_decoded = generator.predict(z_sample)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure)
plt.show()
```

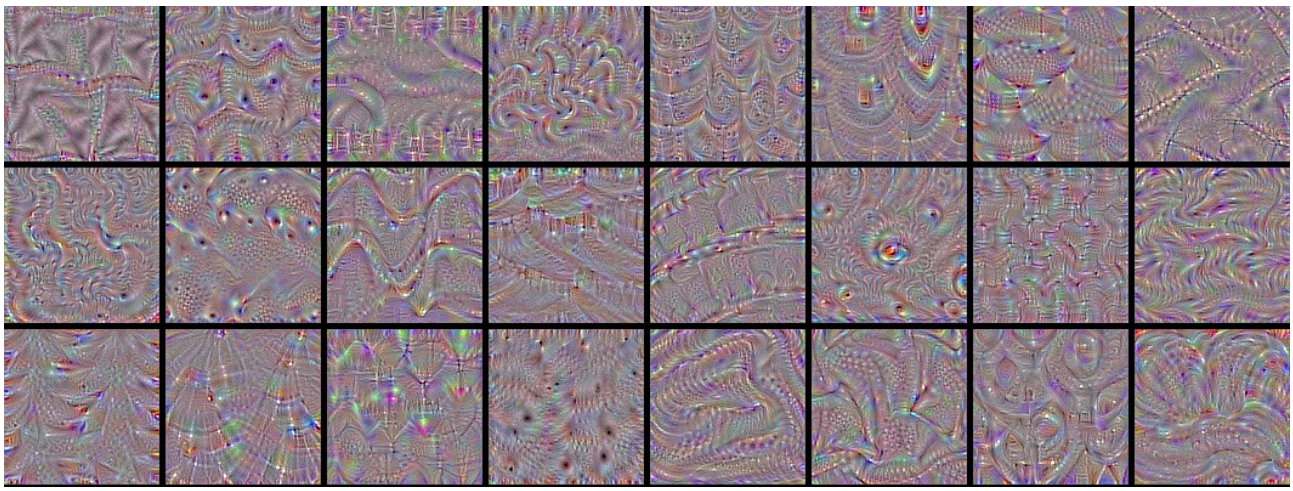


- CNN 眼中的世界：利用 Keras 解释 CNN 的滤波器 本文地址：http://blog.keras.io/how-convolutional-neural-networks-see-the-world.html

本文作者：Francois Chollet

使用 Keras 探索卷积网络的滤波器

本文中我们将利用 Keras 观察 CNN 到底在学些什么，它是如何理解我们送入的训练图片的。我们将使用 Keras 来对滤波器的激活值进行可视化。本文使用的神经网络是 VGG-16，数据集为 ImageNet。本文的代码可以在 github 找到



VGG-16 又称为 OxfordNet，是由牛津视觉几何组（Visual Geometry Group）开发的卷积神经网络结构。该网络赢得了 ILSVR（ImageNet）2014 的冠军。时至今日，VGG 仍然被认为是一个杰出的视觉模型——尽管它的性能实际上已经被后来的 Inception 和 ResNet 超过了。

Lorenzo Baraldi 将 Caffe 预训练好的 VGG16 和 VGG19 模型转化为了 Keras 权重文件，所以我们可以简单的通过载入权重来进行实验。该权重文件可以在这里下载。国内的同学需要自备梯子。（这里是一个网盘保持的 vgg16：http://files.heuritech.com/weights/vgg16_weights.h5 赶紧下载，网盘什么的不知道什么时候就挂了。）

首先，我们在 Keras 中定义 VGG 网络的结构：

```
from keras.models import Sequential
from keras.layers import Convolution2D, ZeroPadding2D, MaxPooling2D

img_width, img_height = 128, 128

# build the VGG16 network
model = Sequential()
model.add(ZeroPadding2D((1, 1), batch_input_shape=(1, 3, img_width,
img_height)))
first_layer = model.layers[-1]
# this is a placeholder tensor that will contain our generated images
input_img = first_layer.input

# build the rest of the network
model.add(Convolution2D(64, 3, 3, activation='relu', name='conv1_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(64, 3, 3, activation='relu', name='conv1_2'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(128, 3, 3, activation='relu', name='conv2_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(128, 3, 3, activation='relu', name='conv2_2'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu', name='conv3_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu', name='conv3_2'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu', name='conv3_3'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))
```

```
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv4_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv4_2'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv4_3'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv5_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv5_2'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv5_3'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

# get the symbolic outputs of each "key" layer (we gave them unique names).
layer_dict = dict([(layer.name, layer) for layer in model.layers])
```

注意我们不需要全连接层，所以网络就定义到最后一个卷积层为止。使用全连接层会将输入大小限制为224×224，即 ImageNet 原图片的大小。这是因为如果输入的图片大小不是 224×224，在从卷积过度到全链接时向量的长度与模型指定的长度不相符。

下面，我们将预训练好的权重载入模型，一般而言我们可以通过`model.load_weights()`载入，但这里我们只载入一部分参数，如果使用该方法的话，模型和参数形式就不匹配了。所以我们需要手工载入：

```
import h5py

weights_path = 'vgg16_weights.h5'

f = h5py.File(weights_path)
for k in range(f.attrs['nb_layers']):
    if k >= len(model.layers):
        # we don't look at the last (fully-connected) layers in the savefile
        break
    g = f['layer_{}'.format(k)]
    weights = [g['param_{}'.format(p)] for p in range(g.attrs['nb_params'])]
    model.layers[k].set_weights(weights)
f.close()
print('Model loaded.')
```

下面，我们要定义一个损失函数，这个损失函数将用于最大化某个指定滤波器的激活值。以该函数为优化目标优化后，我们可以真正看一下使得这个滤波器激活的究竟是些什么东西。

现在我们使用 Keras 的后端来完成这个损失函数，这样这份代码不用修改就可以在 TensorFlow 和 Theano 之间切换了。TensorFlow 在 CPU 上进行卷积要块的多，而目前为止 Theano 在 GPU 上进行卷积要快一些。

```
from keras import backend as K

layer_name = 'conv5_1'
filter_index = 0  # can be any integer from 0 to 511, as there are 512 filters
in that layer

# build a loss function that maximizes the activation
# of the nth filter of the layer considered
layer_output = layer_dict[layer_name].output
```

```
loss = K.mean(layer_output[:, filter_index, :, :])

# compute the gradient of the input picture wrt this loss
grads = K.gradients(loss, input_img)[0]

# normalization trick: we normalize the gradient
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)

# this function returns the loss and grads given the input picture
iterate = K.function([input_img], [loss, grads])
```

注意这里有个小 trick，计算出来的梯度进行了正规化，使得梯度不会过小或过大。这种正规化能够使梯度上升的过程平滑进行。

根据刚刚定义的函数，现在可以对某个滤波器的激活值进行梯度上升。

```
import numpy as np

# we start from a gray image with some noise
input_img_data = np.random.random((1, 3, img_width, img_height)) * 20 + 128.
# run gradient ascent for 20 steps
for i in range(20):
    loss_value, grads_value = iterate([input_img_data])
    input_img_data += grads_value * step
```

使用 TensorFlow 时，这个操作大概只要几秒。

然后我们可以提取出结果，并可视化：

```
from scipy.misc import imsave

# util function to convert a tensor into a valid image
def deprocess_image(x):
    # normalize tensor: center on 0., ensure std is 0.1
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1

    # clip to [0, 1]
    x += 0.5
    x = np.clip(x, 0, 1)

    # convert to RGB array
    x *= 255
    x = x.transpose((1, 2, 0))
    x = np.clip(x, 0, 255).astype('uint8')
    return x

img = input_img_data[0]
img = deprocess_image(img)
imsave('%s_filter_%d.png' % (layer_name, filter_index), img)
```

这里是第 5 卷基层第 0 个滤波器的结果：(pass)

可视化所有的滤波器

下面我们系统的可视化一下各个层的各个滤波器结果，看看 CNN 是如何对输入进行逐层分解的。

第一层的滤波器主要完成方向、颜色的编码，这些颜色和方向与基本的纹理组合，逐渐生成复杂的形状。

可以将每层的滤波器想为基向量，这些基向量一般是过完备的。基向量可以将层的输入紧凑的编码出来。滤波器随着其利用的空域信息的拓宽而更加精细和复杂，(picture pass)

可以观察到，很多滤波器的内容其实是一样的，只不过旋转了一个随机的的角度（如 90 度）而已。这意味着我们可以通过使得卷积滤波器具有旋转不变性而显著减少滤波器的数目，这是一个有趣的研究方向。

令人震惊的是，这种旋转的性质在高层的滤波器中仍然可以被观察到。如 Conv4_1

Deep Dream（nightmare）

另一个有趣的事儿是，如果我们把刚才的随机噪声图片替换为有意义的照片，结果就变的更好玩了。这就是去年由 Google 提出的 Deep Dream。通过选择特定的滤波器组合，我们可以获得一些很有意思的结果。如果你对此感兴趣，可以参考 Keras 的例子 Deep Dream 和 Google 的博客 Google blog post（墙）

愚弄神经网络

如果我们添加上 VGG 的全连接层，然后试图最大化某个指定类别的激活值呢？你会得到一张很像该类别的图片吗？让我们试试。

这种情况下我们的损失函数长这样：

```
layer_output = model.layers[-1].get_output()
loss = K.mean(layer_output[:, output_index])
```

比方说我们来最大化输出下标为 65 的那个类，在 ImageNet 里，这个类是蛇。很快，我们的损失达到了 0.999，即神经网络有 99.9%的概率认为我们生成的图片是一条海蛇，它长这样：

不太像呀，换个类别试试，这次选喜鹊类（第 18 类）

OK，我们的网络认为是喜鹊的东西看起来完全不是喜鹊，往好了说，这个图里跟喜鹊相似的，也不过就是一些局部的纹理，如羽毛，嘴巴之类的。那么，这就意味着卷积神经网络是个很差的工具吗？当然不是，我们按照一个特定任务来训练它，它就会在那个任务上表现的不错。但我们不能有网络“理解”某个概念的错觉。我们不能将网络人格化，它只是工具而已。比如一条狗，它能识别其为狗只是因为它能以很高的概率将其正确分类而已，而不代表它理解关于“狗”的任何外延。

革命尚未成功，同志仍需努力

所以，神经网络到底理解了什么呢？我认为有两件事是它们理解的。

其一，神经网络理解了如何将输入空间解耦为分层次的卷积滤波器组。其二，神经网络理解了从一系列滤波器的组合到一系列特定标签的概率映射。神经网络学习到的东西完全达不到人类的“看见”的意义，从科学的的角度讲，这当然也不意味着我们已经解决了计算机视觉的问题。想得别太多，我们才刚刚踩上计算机视觉天梯的第一步。

有些人说，卷积神经网络学习到的对输入空间的分层次解耦模拟了人类视觉皮层的行为。这种说法可能对也可能不对，但目前未知我们还没有比较强的证据来承认或否认它。当然，有些人可以期望人类的视觉皮层就是以类似的方式学东西的，某种程度上讲，这是对我们视觉世界的自然解耦（就像傅里叶变换

是对周期声音信号的一种解耦一样自然）【译注：这里是说，就像声音信号的傅里叶变换表达了不同频率的声音信号这种很自然很物理的理解一样，我们可能会认为我们对视觉信息的识别就是分层来完成的，圆的是轮子，有四个轮子的是汽车，造型炫酷的汽车是跑车，像这样】。但是，人类对视觉信号的滤波、分层次、处理的本质很可能和我们弱鸡的卷积网络完全不是一回事。视觉皮层不是卷积的，尽管它们也分层，但那些层具有皮质列的结构，而这些结构的真正目的目前还不得而知，这种结构在我们的人工神经网络中还没有出现（尽管乔大帝 Geoff Hinton 正在在这个方面努力）。此外，人类有比给静态图像分类的感知器多得多的视觉感知器，这些感知器是连续而主动的，不是静态而被动的，这些感受器还被如眼动等多种机制复杂控制。

下次有风投或某知名 CEO 警告你要警惕我们深度学习的威胁时，想想上面说的吧。今天我们是有更好的工具来处理复杂的信息了，这很酷，但归根结底它们只是工具，而不是生物。它们做的任何工作在哪个宇宙的标准下都不够格称之为“思考”。在一个石头上画一个笑脸并不会使石头变得“开心”，尽管你的灵长目皮质会告诉你它很开心。

总而言之，卷积神经网络的可视化工作是很让人着迷的，谁能想到仅仅通过简单的梯度下降法和合理的损失函数，加上大规模的数据库，就能学到能很好解释复杂视觉信息的如此漂亮的分层模型呢。深度学习或许在实际的意义上并不智能，但它仍然能够达到几年前任何人都无法达到的效果。现在，如果我们能理解为什么深度学习如此有效，那……嘿嘿:)

- 面向小数据集构建图像分类模型 本文地址：http://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

本文作者：Francois Chollet

概述

在本文中，我们将提供一些面向小数据集（几百张到几千张图片）构造高效、实用的图像分类器的方法。

本文将探讨如下几种方法：

- 从图片中直接训练一个小网络（作为基准方法）
- 利用预训练网络的 bottleneck（瓶颈）特征
- fine-tune 预训练网络的高层

本文需要使用的 Keras 模块有：

- `fit_generator`：用于从 Python 生成器中训练网络
- `ImageDataGenerator`：用于实时数据提升
- 层参数冻结和模型 fine-tune

配置情况

我们的实验基于下面的配置

- 2000 张训练图片构成的数据集，一共两个类别，每类 1000 张

- 安装有 Keras，SciPy，PIL 的机器，如果有 NVIDIA GPU 那就更好了，但因为我们面对的是小数据集，没有也可以。
- 数据集按照下面的形式存放

```
data/
    train/
        dogs/
            dog001.jpg
            dog002.jpg
            ...
        cats/
            cat001/jpg
            cat002.jpg
            ...
    validation/
        dogs/
            dog001.jpg
            dog002.jpg
            ...
        cats/
            cat001/jpg
            cat002.jpg
            ...
```

这份数据集来源于 Kaggle，原数据集有 12500 只猫和 12500 只狗，我们只取了各个类的前 1000 张图片。另外我们还从各个类中取了 400 张额外图片用于测试。

下面是数据集的一些示例图片，图片的数量非常少，这对于图像分类来说是个大麻烦。但现实是，很多真实世界图片获取是很困难的，我们能得到的样本数目确实很有限（比如医学图像，每张正样本都意味着一个承受痛苦的病人:()。对数据科学家而言，我们应该有能够榨取少量数据的全部价值的能力，而不是简单的伸手要更多的数据。

在 Kaggle 的猫狗大战竞赛种，参赛者通过使用现代的深度学习技术达到了 98%的正确率，我们只使用了全部数据的 8%，因此这个问题对我们来说更难。

针对小数据集的深度学习

我经常听到的一种说法是，深度学习只有在你拥有海量数据时才有意义。虽然这种说法并不是完全不对，但却具有较强的误导性。当然，深度学习强调从数据中自动学习特征的能力，没有足够的训练样本，这几乎是不可能的。尤其是当输入的数据维度很高（如图片）时。然而，卷积神经网络作为深度学习的支柱，被设计为针对“感知”问题最好的模型之一（如图像分类问题），即使只有很少的数据，网络也能把特征学的不错。针对小数据集的神经网络依然能够得到合理的结果，并不需要任何手工的特征工程。一言以蔽之，卷积神经网络大法好！

另一方面，深度学习模型天然就具有可重用的特性：比方说，你可以把一个在大规模数据上训练好的图像分类或语音识别的模型重用在另一个很不一样的问题上，而只需要做有限的一点改动。尤其在计算机视觉领域，许多预训练的模型现在都被公开下载，并被重用在其他问题上以提升在小数据集上的性能。

数据预处理与数据提升

为了尽量利用我们有限的训练数据，我们将通过一系列随机变换堆数据进行提升，这样我们的模型将看不到任何两张完全相同的图片，这有利于我们抑制过拟合，使得模型的泛化能力更好。

在 Keras 中，这个步骤可以通过 `keras.preprocessing.image.ImageGenerator` 来实现，这个类使你可以：

- 在训练过程中，设置要施行的随机变换
- 通过 `.flow` 或 `.flow_from_directory(directory)` 方法实例化一个针对图像 batch 的生成器，这些生成器可以被用作 keras 模型相关方法的输入，如 `fit_generator`，`evaluate_generator` 和 `predict_generator`

现在让我们看个例子：

```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')
```

上面显示的只是一部分选项，请阅读文档的相关部分来查看全部可用的选项。我们来快速的浏览一下这些选项的含义：

- `rotation_range` 是一个 0~180 的度数，用来指定随机选择图片的角度。
- `width_shift` 和 `height_shift` 用来指定水平和竖直方向随机移动的程度，这是两个 0~1 之间的比例。
- `rescale` 值将在执行其他处理前乘到整个图像上，我们的图像在 RGB 通道都是 0~255 的整数，这样的操作可能使图像的值过高或过低，所以我们将这个值定为 0~1 之间的数。

- shear_range 是用来进行剪切变换的程度，参考剪切变换
- zoom_range 用来进行随机的放大
- horizontal_flip 随机的对图片进行水平翻转，这个参数适用于水平翻转不影响图片语义的时候
- fill_mode 用来指定当需要进行像素填充，如旋转，水平和竖直位移时，如何填充新出现的像素

下面我们使用这个工具来生成图片，并将它们保存在一个临时文件夹中，这样我们可以感觉一下数据提升究竟做了什么事。为了使图片能够展示出来，这里没有使用 rescaling

```
from keras.preprocessing.image import ImageDataGenerator, array_to_img,
img_to_array, load_img

datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')

img = load_img('data/train/cats/cat.0.jpg')  # this is a PIL image
x = img_to_array(img)  # this is a Numpy array with shape (3, 150, 150)
x = x.reshape((1,) + x.shape)  # this is a Numpy array with shape (1, 3, 150,
150)

# the .flow() command below generates batches of randomly transformed images
# and saves the results to the `preview/` directory
i = 0
for batch in datagen.flow(x, batch_size=1,
                          save_to_dir='preview', save_prefix='cat',
save_format='jpeg'):
    i += 1
    if i > 20:
        break  # otherwise the generator would loop indefinitely
```

下面是一张图片被提升以后得到的多个结果：

在小数据集上训练神经网络：40 行代码达到 80%的准确率

进行图像分类的正确工具是卷积网络，所以我们来试试用卷积神经网络搭建一个初级的模型。因为我们的样本数很少，所以我们应该对过拟合的问题多加注意。当一个模型从很少的样本中学习到不能推广到新数据的模式时，我们称为出现了过拟合的问题。过拟合发生时，模型试图使用不相关的特征来进行预测。例如，你有三张伐木工人的照片，有三张水手的照片。六张照片中只有一个伐木工人戴了帽子，如果你认为戴帽子是能将伐木工人与水手区别开的特征，那么此时你就是一个差劲的分类器。

数据提升是对抗过拟合问题的一个武器，但还不够，因为提升过的数据仍然是高度相关的。对抗过拟合的你应该主要关注的是模型的“熵容量”——模型允许存储的信息量。能够存储更多信息的模型能够利用更多的特征取得更好的性能，但也有存储不相关特征的风险。另一方面，只能存储少量信息的模型会将存储的特征主要集中在真正相关的特征上，并有更好的泛化性能。

有很多不同的方法来调整模型的“熵容量”，常见的一种选择是调整模型的参数数目，即模型的层数和每层的规模。另一种方法是对权重进行正则化约束，如 L1 或 L2.这种约束会使模型的权重偏向较小的值。

在我们的模型里，我们使用了很小的卷积网络，只有很少的几层，每层的滤波器数目也不多。再加上数据提升和 Dropout，就差不多了。Dropout 通过防止一层看到两次完全一样的模式来防止过拟合，相当于也是一种数据提升的方法。（你可以说 dropout 和数据提升都在随机扰乱数据的相关性）

下面展示的代码是我们的第一个模型，一个很简单的 3 层卷积加上 ReLU 激活函数，再接 max-pooling 层。这个结构和 Yann LeCun 在 1990 年发布的图像分类器很相似（除了 ReLU）

这个实验的全部代码在这里

```
from keras.models import Sequential
from keras.layers import Convolution2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense

model = Sequential()
model.add(Convolution2D(32, 3, 3, input_shape=(3, 150, 150)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(32, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# the model so far outputs 3D feature maps (height, width, features)
```

然后我们接了两个全连接网络，并以单个神经元和 sigmoid 激活结束模型。这种选择会产生二分类的结果，与这种配置相适应，我们使用 `binary_crossentropy` 作为损失函数。

```
model.add(Flatten())  # this converts our 3D feature maps to 1D feature vectors
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy',
```

```
              optimizer='rmsprop',
              metrics=['accuracy'])
```

然后我们开始准备数据，使用.flow from directory()来从我们的jpgs图片中直接产生数据和标签。

```
# this is the augmentation configuration we will use for training
train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)

# this is the augmentation configuration we will use for testing:
# only rescaling
test_datagen = ImageDataGenerator(rescale=1./255)

# this is a generator that will read pictures found in
# subfolers of 'data/train', and indefinitely generate
# batches of augmented image data
train_generator = train_datagen.flow_from_directory(
        'data/train',  # this is the target directory
        target_size=(150, 150),  # all images will be resized to 150x150
        batch_size=32,
        class_mode='binary')  # since we use binary_crossentropy loss, we need
binary labels

# this is a similar generator, for validation data
validation_generator = test_datagen.flow_from_directory(
        'data/validation',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')
```

然后我们可以用这个生成器来训练网络了，在GPU上每个epoch耗时20~30秒，在CPU上耗时300~400秒，所以如果你不是很着急，在CPU上跑这个模型也是完全可以的。

```
model.fit_generator(
        train_generator,
        samples_per_epoch=2000,
        nb_epoch=50,
        validation_data=validation_generator,
        nb_val_samples=800)
model.save_weights('first_try.h5')  # always save your weights after training or
during training
```

这个模型在50个epoch后的准确率为79%~81%，别忘了我们只用了8%的数据，也没有花时间来做模型和超参数的优化。在Kaggle中，这个模型已经可以进前100名了（一共215队参与），估计剩下的115队都没有用深度学习:)

注意这个准确率的变化可能会比较大，因为准确率本来就是一个变化较高的评估参数，而且我们只有800个样本用来测试。比较好的验证方法是使用K折交叉验证，但每轮验证中我们都要训练一个模型。

使用预训练网络的bottleneck特征：一分钟达到90%的正确率

一个稍微讲究一点的办法是，利用在大规模数据集上预训练好的网络。这样的网络在多数的计算机视觉问题上都能取得不错的特征，利用这样的特征可以让我们获得更高的准确率。

我们将使用 vgg-16 网络，该网络在 ImageNet 数据集上进行训练，这个模型我们之前提到过了。因为 ImageNet 数据集包含多种“猫”类和多种“狗”类，这个模型已经能够学习与我们这个数据集相关的特征了。事实上，简单的记录原来网络的输出而不用 bottleneck 特征就已经足够把我们的问题解决的不错了。不过我们这里讲的方法对其他的类似问题有更好的推广性，包括在 ImageNet 中没有出现的类别的分类问题。

VGG-16 的网络结构如下：(picture)

我们的方法是这样的，我们将利用网络的卷积层部分，把全连接以上的部分抛掉。然后在我们的训练集和测试集上跑一遍，将得到的输出（即“bottleneck feature”，网络在全连接之前的最后一层激活的 feature map）记录在两个 numpy array 里。然后我们基于记录下来的特征训练一个全连接网络。

我们将这些特征保存为离线形式，而不是将我们的全连接模型直接加到网络上并冻结之前的层参数进行训练的原因是处于计算效率的考虑。运行 VGG 网络的代价是非常高昂的，尤其是在 CPU 上运行，所以我们只想运行一次。这也是我们不进行数据提升的原因。

我们不再赘述如何搭建 vgg-16 网络了，这件事之前已经说过，在 keras 的 example 里也可以找到。但让我们看看如何记录 bottleneck 特征。

```
generator = datagen.flow_from_directory(

        'data/train',
        target_size=(150, 150),
        batch_size=32,
        class_mode=None,  # this means our generator will only yield batches of
data, no labels
        shuffle=False)  # our data will be in order, so all first 1000 images
will be cats, then 1000 dogs
# the predict_generator method returns the output of a model, given
# a generator that yields batches of numpy data
bottleneck_features_train = model.predict_generator(generator, 2000)
# save the output as a Numpy array
np.save(open('bottleneck_features_train.npy', 'w'), bottleneck_features_train)

generator = datagen.flow_from_directory(
        'data/validation',
        target_size=(150, 150),
        batch_size=32,
        class_mode=None,
        shuffle=False)
bottleneck_features_validation = model.predict_generator(generator, 800)
np.save(open('bottleneck_features_validation.npy', 'w'),
bottleneck_features_validation)
```

记录完毕后我们可以将数据载入，用于训练我们的全连接网络：

```
train_data = np.load(open('bottleneck_features_train.npy'))

# the features were saved in order, so recreating the labels is easy
train_labels = np.array([0] * 1000 + [1] * 1000)

validation_data = np.load(open('bottleneck_features_validation.npy'))
validation_labels = np.array([0] * 400 + [1] * 400)

model = Sequential()
```

```
model.add(Flatten(input_shape=train_data.shape[1:]))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

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

model.fit(train_data, train_labels,
          nb_epoch=50, batch_size=32,
          validation_data=(validation_data, validation_labels))
model.save_weights('bottleneck_fc_model.h5')
```

因为特征的 size 很小，模型在 CPU 上跑的也会很快，大概 1s 一个 epoch，最后我们的准确率是 90%~91%，这么好的结果多半归功于预训练的 vgg 网络帮助我们提取特征。

在预训练的网络上 fine-tune

为了进一步提高之前的结果，我们可以试着 fine-tune 网络的后面几层。Fine-tune 以一个预训练好的网络为基础，在新的数据集上重新训练一小部分权重。在这个实验中，fine-tune 分三个步骤

- 搭建 vgg-16 并载入权重
- 将之前定义的全连接网络加在模型的顶部，并载入权重
- 冻结 vgg16 网络的一部分参数 (picture after next page)

注意：

- 为了进行 fine-tune,所有的层都应该以训练好的权重为初始值，例如，你不能将随机初始的全连接放在预训练的卷积层之上，这是因为由随机权重产生的大梯度将会破坏卷积层预训练的权重。在我们的情形中，这就是为什么我们首先训练顶层分类器，然后再基于它进行 fine-tune 的原因
- 我们选择只 fine-tune 最后的卷积块，而不是整个网络，这是为了防止过拟合。整个网络具有巨大的熵容量，因此具有很高的过拟合倾向。由底层卷积模块学习到的特征更加一般，更加不具有抽象性，因此我们要保持前两个卷积块（学习一般特征）不动，只 fine-tune 后面的卷积块（学习特别的特征）。
- fine-tune 应该在很低的学习率下进行，通常使用 SGD 优化而不是其他自适应学习率的优化算法，如 RMSProp。这是为了保证更新的幅度保持在较低的程度，以免毁坏预训练的特征。

代码如下，首先在初始化好的 vgg 网络上添加我们预训练好的模型：

```
# build a classifier model to put on top of the convolutional model
top_model = Sequential()
top_model.add(Flatten(input_shape=model.output_shape[1:]))
top_model.add(Dense(256, activation='relu'))
top_model.add(Dropout(0.5))
top_model.add(Dense(1, activation='sigmoid'))

# note that it is necessary to start with a fully-trained
# classifier, including the top classifier,
# in order to successfully do fine-tuning
top_model.load_weights(top_model_weights_path)
```

```
# add the model on top of the convolutional base
model.add(top_model)
```

然后将最后一个卷积块前的卷积层参数冻结：

```
# set the first 25 layers (up to the last conv block)
# to non-trainable (weights will not be updated)
for layer in model.layers[:25]:
    layer.trainable = False

# compile the model with a SGD/momentum optimizer
# and a very slow learning rate.
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])
```

然后以很低的学习率进行训练：

```
# prepare data augmentation configuration
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(
        train_data_dir,
        target_size=(img_height, img_width),
        batch_size=32,
        class_mode='binary')

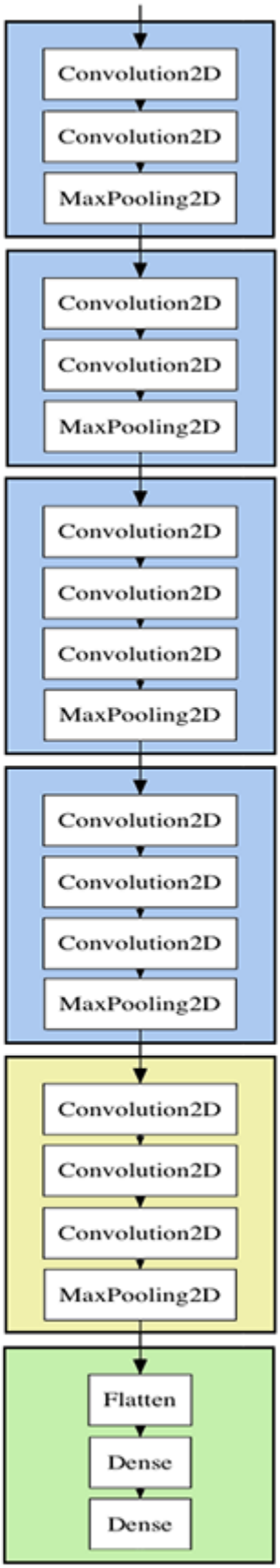
validation_generator = test_datagen.flow_from_directory(
        validation_data_dir,
        target_size=(img_height, img_width),
        batch_size=32,
        class_mode='binary')

# fine-tune the model
model.fit_generator(
        train_generator,
        samples_per_epoch=nb_train_samples,
        nb_epoch=nb_epoch,
        validation_data=validation_generator,
        nb_val_samples=nb_validation_samples)
```

在 50 个 epoch 之后该方法的准确率为 94%，非常成功

通过下面的方法你可以达到 95%以上的正确率：

- 更加强烈的数据提升
- 更加强烈的 dropout
- 使用 L1 和 L2 正则项（也称为权重衰减）
- fine-tune 更多的卷积块（配合更大的正则）



Conv block 1: frozen

Conv block 2: frozen

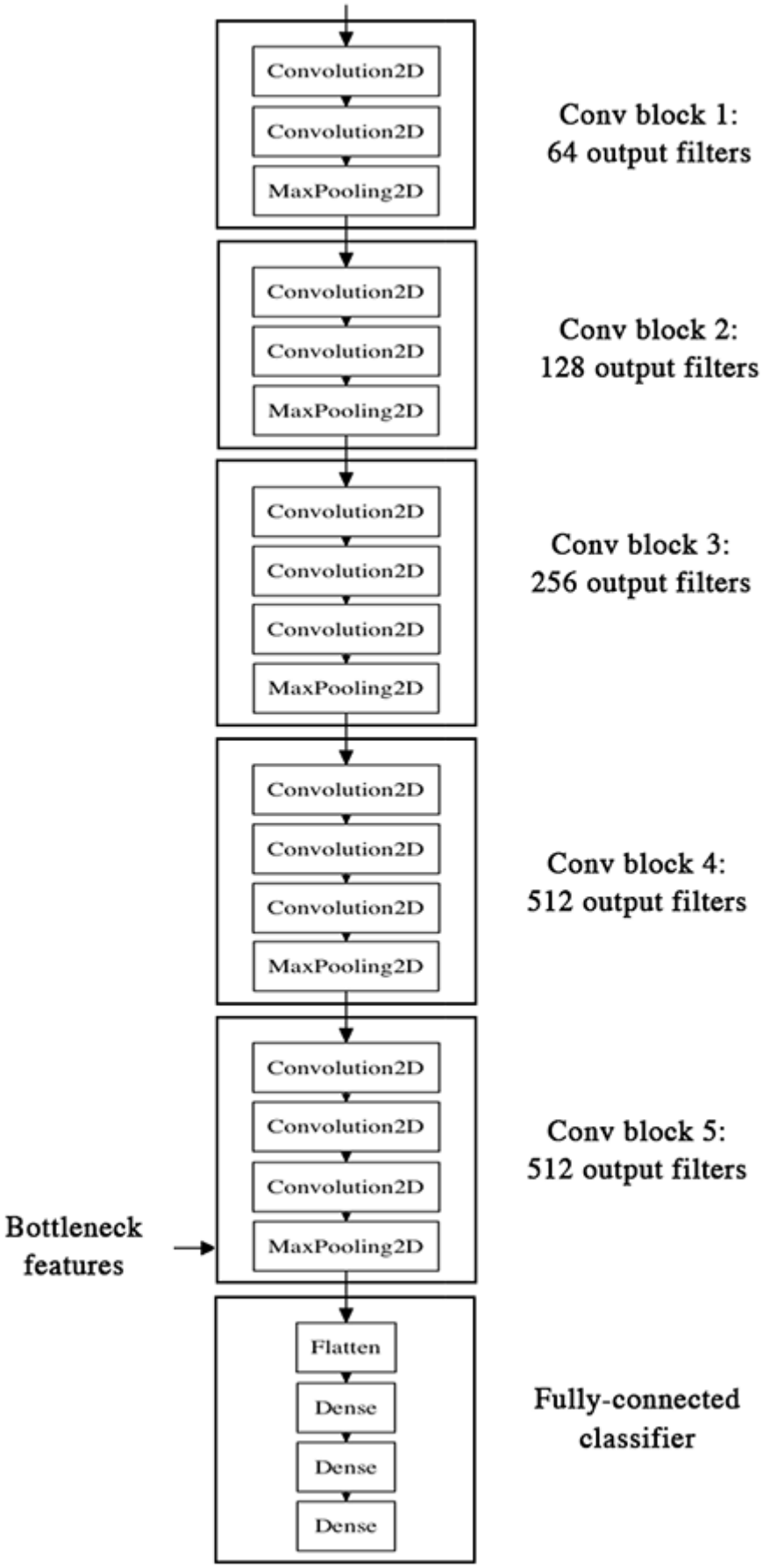
Conv block 3: frozen

Conv block 4: frozen

We fine-tune Conv block 5

We fine-tune our own fully-connected classifier

•


Convolution2D
Convolution2D
MaxPooling2D
Conv block 1:
64 output filters
Convolution2D
Convolution2D
MaxPooling2D
Conv block 2:
128 output filters
Convolution2D
Convolution2D
Convolution2D
MaxPooling2D
Conv block 3:
256 output filters
Convolution2D
Convolution2D
Convolution2D
MaxPooling2D
Conv block 4:
512 output filters
Convolution2D
Convolution2D
Convolution2D
MaxPooling2D
Conv block 5:
512 output filters
Bottleneck
features
Flatten
Dense
Dense
Dense
Fully-connected
classifier

- 将 Keras 作为 tensorflow 的精简接口

使用 Keras 作为 TensorFlow 工作流的一部分

如果 Tensorflow 是你的首选框架，并且你想找一个简化的、高层的模型定义接口来让自己活的不那么累，那么这篇文章就是给你看的

Keras 的层和模型与纯 TensorFlow 的 tensor 完全兼容，因此，Keras 可以作为 TensorFlow 的模型定义，甚至可以与其他 TensoFlow 库协同工作。

注意，本文假定你已经把 Keras 配置为 tensorflow 后端，如果你不懂怎么配置，请查看这里

在 tensorflow 中调用 Keras 层

让我们以一个简单的例子开始：MNIST 数字分类。我们将以 Keras 的全连接层堆叠构造一个 TensorFlow 的分类器，

```
import tensorflow as tf
sess = tf.Session()

from keras import backend as K
K.set_session(sess)
```

然后，我们开始用 tensorflow 构建模型：

```
# this placeholder will contain our input digits, as flat vectors
img = tf.placeholder(tf.float32, shape=(None, 784))
```

用 Keras 可以加速模型的定义过程：

```
from keras.layers import Dense

# Keras layers can be called on TensorFlow tensors:
x = Dense(128, activation='relu')(img)  # fully-connected layer with 128 units
and ReLU activation
x = Dense(128, activation='relu')(x)
preds = Dense(10, activation='softmax')(x)  # output layer with 10 units and a
softmax activation
```

定义标签的占位符和损失函数：

```
labels = tf.placeholder(tf.float32, shape=(None, 10))

from keras.objectives import categorical_crossentropy
loss = tf.reduce_mean(categorical_crossentropy(labels, preds))
```

然后，我们可以用 tensorflow 的优化器来训练模型：

```
from tensorflow.examples.tutorials.mnist import input_data
mnist_data = input_data.read_data_sets('MNIST_data', one_hot=True)

train_step = tf.train.GradientDescentOptimizer(0.5).minimize(loss)
with sess.as_default():
    for i in range(100):
        batch = mnist_data.train.next_batch(50)
        train_step.run(feed_dict={img: batch[0],
                                  labels: batch[1]})
```

最后我们来评估一下模型性能：

```
from keras.metrics import categorical_accuracy as accuracy

acc_value = accuracy(labels, preds)
with sess.as_default():
    print acc_value.eval(feed_dict={img: mnist_data.test.images,
                                    labels: mnist_data.test.labels})
```

我们只是将 Keras 作为生成从 tensor 到 tensor 的函数（op）的快捷方法而已，优化过程完全采用的原生 tensorflow 的优化器，而不是 Keras 优化器，我们压根不需要 Keras 的 Model

关于原生 TensorFlow 和 Keras 的优化器的一点注记：虽然有点反直觉，但 Keras 的优化器要比 TensorFlow 的优化器快大概 5-10%。虽然这种速度的差异基本上没什么差别。

训练和测试行为不同

有些 Keras 层，如 BN，Dropout，在训练和测试过程中的行为不一致，你可以通过打印 layer.uses_learning_phase 来确定当前层工作在训练模式还是测试模式。

如果你的模型包含这样的层，你需要指定你希望模型工作在什么模式下，通过 Keras 的 backend 你可以了解当前的工作模式：

```
from keras import backend as K
print K.learning_phase()
```

向 feed_dict 中传递 1（训练模式）或 0（测试模式）即可指定当前工作模式：

```
# train mode
train_step.run(feed_dict={x: batch[0], labels: batch[1], K.learning_phase(): 1})
```

例如，下面代码示范了如何将 Dropout 层加入刚才的模型中：

```
from keras.layers import Dropout
from keras import backend as K

img = tf.placeholder(tf.float32, shape=(None, 784))
labels = tf.placeholder(tf.float32, shape=(None, 10))

x = Dense(128, activation='relu')(img)
x = Dropout(0.5)(x)
x = Dense(128, activation='relu')(x)
x = Dropout(0.5)(x)
preds = Dense(10, activation='softmax')(x)

loss = tf.reduce_mean(categorical_crossentropy(labels, preds))

train_step = tf.train.GradientDescentOptimizer(0.5).minimize(loss)
with sess.as_default():
    for i in range(100):
        batch = mnist_data.train.next_batch(50)
        train_step.run(feed_dict={img: batch[0],
                                  labels: batch[1],
                                  K.learning_phase(): 1})

acc_value = accuracy(labels, preds)
with sess.as_default():
    print acc_value.eval(feed_dict={img: mnist_data.test.images,
                                    labels: mnist_data.test.labels,
```

```
                                    K.learning_phase(): 0})
```

与变量名作用域和设备作用域的兼容

Keras 的层与模型和 tensorflow 的命名完全兼容，例如：

```
x = tf.placeholder(tf.float32, shape=(None, 20, 64))
with tf.name_scope('block1'):
    y = LSTM(32, name='mylstm')(x)
```

我们 LSTM 层的权重将会被命名为 block1/mylstm_W_i, block1/mylstm_U, 等.. 类似的，设备的命名也会像你期望的一样工作：

```
with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32, shape=(None, 20, 64))
    y = LSTM(32)(x)  # all ops / variables in the LSTM layer will live on GPU:0
```

与 Graph 的作用域兼容

任何在 tensorflow 的 Graph 作用域定义的 Keras 层或模型的所有变量和操作将被生成为该 Graph 的一个部分，例如，下面的代码将会以你所期望的形式工作

```
from keras.layers import LSTM
import tensorflow as tf

my_graph = tf.Graph()
with my_graph.as_default():
    x = tf.placeholder(tf.float32, shape=(None, 20, 64))
    y = LSTM(32)(x)  # all ops / variables in the LSTM layer are created as part
of our graph
```

与变量作用域兼容

变量共享应通过多次调用同样的 Keras 层或模型来实现，而不是通过 TensorFlow 的变量作用域实现。TensorFlow 变量作用域将对 Keras 层或模型没有任何影响。更多 Keras 权重共享的信息请参考这里

Keras 通过重用相同层或模型的对象来完成权值共享，这是一个例子：

```
# instantiate a Keras layer
lstm = LSTM(32)

# instantiate two TF placeholders
x = tf.placeholder(tf.float32, shape=(None, 20, 64))
y = tf.placeholder(tf.float32, shape=(None, 20, 64))

# encode the two tensors with the *same* LSTM weights
x_encoded = lstm(x)
y_encoded = lstm(y)
```

收集可训练权重与状态更新

某些 Keras 层，如状态 RNN 和 BN 层，其内部的更新需要作为训练过程的一步来进行，这些更新被存储在一个 tensor tuple 里：layer.updates，你应该生成 assign 操作来使在训练的每一步这些更新能够被运行，这里是例子：

```
from keras.layers import BatchNormalization
```

```
layer = BatchNormalization()(x)

update_ops = []
for old_value, new_value in layer.updates:
    update_ops.append(tf.assign(old_value, new_value))
```

注意如果你使用 Keras 模型，model.updates 将与上面的代码作用相同（收集模型中所有更新）

另外，如果你需要显式的收集一个层的可训练权重，你可以通过 layer.trainable_weights 来实现，对模型而言是 model.trainable_weights，它是一个 tensorflow 变量对象的列表：

```
from keras.layers import Dense

layer = Dense(32)(x)  # instantiate and call a layer
print layer.trainable_weights  # list of TensorFlow Variables
```

这些东西允许你实现你基于 TensorFlow 优化器实现自己的训练程序

使用 Keras 模型与 TensorFlow 协作

将 Keras Sequential 模型转换到 TensorFlow 中

假如你已经有一个训练好的 Keras 模型，如 VGG-16，现在你想将它应用在你的 TensorFlow 工作中，应该怎么办？

首先，注意如果你的预训练权重含有使用 Theano 训练的卷积层的话，你需要对这些权重的卷积核进行转换，这是因为 Theano 和 TensorFlow 对卷积的实现不同，TensorFlow 和 Caffe 实际上实现的是相关性计算。点击这里查看详细示例。

假设你从下面的 Keras 模型开始，并希望对其进行修改以使得它可以以一个特定的 tensorflow 张量 my_input_tensor 为输入，这个 tensor 可能是一个数据 feeder 或别的 tensorflow 模型的输出

```
# this is our initial Keras model
model = Sequential()
first_layer = Dense(32, activation='relu', input_dim=784)
model.add(Dense(10, activation='softmax'))
```

你只需要在实例化该模型后，使用 set_input 来修改首层的输入，然后将剩下模型搭建于其上：

```
# this is our modified Keras model
model = Sequential()
first_layer = Dense(32, activation='relu', input_dim=784)
first_layer.set_input(my_input_tensor)

# build the rest of the model as before
model.add(first_layer)
model.add(Dense(10, activation='softmax'))
```

在这个阶段，你可以调用 model.load_weights(weights_file)来加载预训练的权重

然后，你或许会收集该模型的输出张量：

```
output_tensor = model.output
```

对 TensorFlow 张量中调用 Keras 模型

Keras 模型与 Keras 层的行为一致，因此可以被调用于 TensorFlow 张量上：

```
from keras.models import Sequential

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=784))
model.add(Dense(10, activation='softmax'))

# this works!
x = tf.placeholder(tf.float32, shape=(None, 784))
y = model(x)
```

注意，调用模型时你同时使用了模型的结构与权重，当你在一个 tensor 上调用模型时，你就在该 tensor 上创造了一些操作，这些操作重用了已经在模型中出现的 TensorFlow 变量的对象

多 GPU 和分布式训练

将 Keras 模型分散在多个 GPU 中训练

TensorFlow 的设备作用域完全与 Keras 的层和模型兼容，因此你可以使用它们来将一个图的特定部分放在不同的 GPU 中训练，这里是一个简单的例子：

```
with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32, shape=(None, 20, 64))
    y = LSTM(32)(x)  # all ops in the LSTM layer will live on GPU:0

with tf.device('/gpu:1'):
    x = tf.placeholder(tf.float32, shape=(None, 20, 64))
    y = LSTM(32)(x)  # all ops in the LSTM layer will live on GPU:1
```

注意，由 LSTM 层创建的变量将不会生存在 GPU 上，不管 TensorFlow 变量在哪里创建，它们总是生存在 CPU 上，TensorFlow 将隐含的处理设备之间的转换

如果你想在多个 GPU 上训练同一个模型的多个副本，并在多个副本中进行权重共享，首先你应该在一个设备作用域下实例化你的模型或层，然后在不同 GPU 设备的作用域下多次调用该模型实例，如：

```
with tf.device('/cpu:0'):
    x = tf.placeholder(tf.float32, shape=(None, 784))

    # shared model living on CPU:0
    # it won't actually be run during training; it acts as an op template
    # and as a repository for shared variables
    model = Sequential()
    model.add(Dense(32, activation='relu', input_dim=784))
    model.add(Dense(10, activation='softmax'))

# replica 0
with tf.device('/gpu:0'):
    output_0 = model(x)  # all ops in the replica will live on GPU:0

# replica 1
with tf.device('/gpu:1'):
    output_1 = model(x)  # all ops in the replica will live on GPU:1

# merge outputs on CPU
with tf.device('/cpu:0'):
```

```
    preds = 0.5 * (output_0 + output_1)

# we only run the `preds` tensor, so that only the two
# replicas on GPU get run (plus the merge op on CPU)
output_value = sess.run([preds], feed_dict={x: data})
```

分布式训练

通过注册 Keras 会话到一个集群上，你可以简单的实现分布式训练：

```
server = tf.train.Server.create_local_server()
sess = tf.Session(server.target)

from keras import backend as K
K.set_session(sess)
```

关于 TensorFlow 进行分布式训练的配置信息，请参考这里

使用 TensorFlow-serving 导出模型

TensorFlow-Serving 是由 Google 开发的用于将 TensoFlow 模型部署于生产环境的工具

任何 Keras 模型都可以被 TensorFlow-serving 所导出（只要它只含有一个输入和一个输出，这是 TF-serving 的限制），不管它是否作为 TensroFlow 工作流的一部分。事实上你甚至可以使用 Theano 训练你的 Keras 模型，然后将其切换到 tensorflow 后端，然后导出模型

如果你的 graph 使用了 Keras 的 learning phase（在训练和测试中行为不同），你首先要做的事就是在 graph 中硬编码你的工作模式（设为 0，即测试模式），该工作通过 1）使用 Keras 的后端注册一个 learning phase 常量，2）重新构建模型，来完成。

这里是实践中的示范：

```
from keras import backend as K

K.set_learning_phase(0)  # all new operations will be in test mode from now on

# serialize the model and get its weights, for quick re-building
config = previous_model.get_config()
weights = previous_model.get_weights()

# re-build a model where the learning phase is now hard-coded to 0
from keras.models import model_from_config
new_model = model_from_config(config)
new_model.set_weights(weights)
```

现在，我们可使用 Tensorflow-serving 来导出模型，按照官方教程的指导：

```
from tensorflow_serving.session_bundle import exporter

export_path = ... # where to save the exported graph
export_version = ... # version number (integer)

saver = tf.train.Saver(sharded=True)
model_exporter = exporter.Exporter(saver)
signature = exporter.classification_signature(input_tensor=model.input,
                                              scores_tensor=model.output)
model_exporter.init(sess.graph.as_graph_def(),
                    default_graph_signature=signature)
```

```
model_exporter.export(export_path, tf.constant(export_version), sess)
```

如想看到包含本教程的新主题，请看我的 Twitter

- 在 Keras 模型中使用预训练的词向量 本文地址：http://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html

本文作者：Francois Chollet

什么是词向量?

”词向量”（词嵌入）是将一类将词的语义映射到向量空间中去的自然语言处理技术。即将一个词用特定的向量来表示，向量之间的距离（例如，任意两个向量之间的 L2 范式距离或更常用的余弦距离）一定程度上表征了的词之间的语义关系。由这些向量形成的几何空间被称为一个嵌入空间。

例如，“椰子”和“北极熊”是语义上完全不同的词，所以它们的词向量在一个合理的嵌入空间的距离将会非常遥远。但“厨房”和“晚餐”是相关的话，所以它们的词向量之间的距离会相对小。

理想的情况下，在一个良好的嵌入空间里，从“厨房”向量到“晚餐”向量的“路径”向量会精确地捕捉这两个概念之间的语义关系。在这种情况下，“路径”向量表示的是“发生的地点”，所以你会期望“厨房”向量 - “晚餐"向量（两个词向量的差异）捕捉到“发生的地点”这样的语义关系。基本上，我们应该有向量等式：晚餐 + 发生的地点 = 厨房（至少接近）。如果真的是这样的话，那么我们可以使用这样的关系向量来回答某些问题。例如，应用这种语义关系到一个新的向量，比如“工作”，我们应该得到一个有意义的等式，工作+ 发生的地点 = 办公室，来回答“工作发生在哪里？”。

词向量通过降维技术表征文本数据集中的词的共现信息。方法包括神经网络(“Word2vec”技术)，或矩阵分解。

GloVe 词向量

本文使用 GloVe 词向量。GloVe 是 "Global Vectors for Word Representation"的缩写，一种基于共现矩阵分解的词向量。本文所使用的 GloVe 词向量是在 2014 年的英文维基百科上训练的，有 400k 个不同的词，每个词用 100 维向量表示。点此下载 (友情提示，词向量文件大小约为 822M)

---

20 Newsgroup dataset

本文使用的数据集是著名的"20 Newsgroup dataset"。该数据集共有 20 种新闻文本数据，我们将实现对该数据集的文本分类任务。数据集的说明和下载请参考这里。

不同类别的新闻包含大量不同的单词，在语义上存在极大的差别，。一些新闻类别如下所示

comp.sys.ibm.pc.hardware comp.graphics comp.os.ms-windows.misc
comp.sys.mac.hardware comp.windows.x rec.autos rec.motorcycles rec.sport.baseball

rec.sport.hockey

实验方法

以下是我们如何解决分类问题的步骤

- 将所有的新闻样本转化为词索引序列。所谓词索引就是为每一个词依次分配一个整数 ID。遍历所有的新闻文本，我们只保留最参见的 20,000 个词，而且 每个新闻文本最多保留 1000 个词。
- 生成一个词向量矩阵。第 i 列表示词索引为 i 的词的词向量。
- 将词向量矩阵载入 Keras Embedding 层，设置该层的权重不可再训练（也就是说在之后的网络训练过程中，词向量不再改变）。
- Keras Embedding 层之后连接一个 1D 的卷积层，并用一个 softmax 全连接输出新闻类别

数据预处理

我们首先遍历下语料文件下的所有文件夹，获得不同类别的新闻以及对应的类别标签，代码如下所示

```
texts = []  # list of text samples
labels_index = {}  # dictionary mapping label name to numeric id
labels = []  # list of label ids
for name in sorted(os.listdir(TEXT_DATA_DIR)):
    path = os.path.join(TEXT_DATA_DIR, name)
    if os.path.isdir(path):
        label_id = len(labels_index)
        labels_index[name] = label_id
        for fname in sorted(os.listdir(path)):
            if fname.isdigit():
                fpath = os.path.join(path, fname)
                f = open(fpath)
                texts.append(f.read())
                f.close()
                labels.append(label_id)

print('Found %s texts.' % len(texts))
```

之后，我们可以新闻样本转化为神经网络训练所用的张量。所用到的 Keras 库是 keras.preprocessing.text.Tokenizer 和 keras.preprocessing.sequence.pad_sequences。代码如下所示

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

tokenizer = Tokenizer(nb_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)

labels = to_categorical(np.asarray(labels))
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)

# split the data into a training set and a validation set
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]
nb_validation_samples = int(VALIDATION_SPLIT * data.shape[0])
```

```
x_train = data[:-nb_validation_samples]
y_train = labels[:-nb_validation_samples]
x_val = data[-nb_validation_samples:]
y_val = labels[-nb_validation_samples:]
```

Embedding layer 设置

接下来，我们从 GloVe 文件中解析出每个词和它所对应的词向量，并用字典的方式存储

```
embeddings_index = {}
f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))
```

此时，我们可以根据得到的字典生成上文所定义的词向量矩阵

```
embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
```

现在我们将这个词向量矩阵加载到 Embedding 层中，注意，我们设置 trainable=False 使得这个编码层不可再训练。

```
from keras.layers import Embedding

embedding_layer = Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=False)
```

一个 Embedding 层的输入应该是一系列的整数序列，比如一个 2D 的输入，它的 shape 值为(samples, indices)，也就是一个 samples 行，indeces 列的矩阵。每一次的 batch 训练的输入应该被 padded 成相同大小（尽管 Embedding 层有能力处理不定长序列，如果你不指定数列长度这一参数） dim). 所有的序列中的整数都将被对应的词向量矩阵中对应的列（也就是它的词向量）代替,比如序列[1,2]将被序列[词向量[1],词向量[2]]代替。这样，输入一个 2D 张量后，我们可以得到一个 3D 张量。

训练 1D 卷积

最后，我们可以使用一个小型的 1D 卷积解决这个新闻分类问题。

```
sequence_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences = embedding_layer(sequence_input)
x = Conv1D(128, 5, activation='relu')(embedded_sequences)
x = MaxPooling1D(5)(x)
x = Conv1D(128, 5, activation='relu')(x)
```

```
x = MaxPooling1D(5)(x)
x = Conv1D(128, 5, activation='relu')(x)
x = MaxPooling1D(35)(x)  # global max pooling
x = Flatten()(x)
x = Dense(128, activation='relu')(x)
preds = Dense(len(labels_index), activation='softmax')(x)

model = Model(sequence_input, preds)
model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['acc'])

# happy learning!
model.fit(x_train, y_train, validation_data=(x_val, y_val),
          nb_epoch=2, batch_size=128)
```

在两次迭代之后，这个模型最后可以达到 0.95 的分类准确率（4:1 分割训练和测试集合）。你可以利用正则方法（例如 dropout）或在 Embedding 层上进行 fine-tuning 获得更高的准确率。

我们可以做一个对比实验，直接使用 Keras 自带的 Embedding 层训练词向量而不用 GloVe 向量。代码如下所示

```
embedding_layer = Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            input_length=MAX_SEQUENCE_LENGTH)
```

两次迭代之后，我们可以得到 0.9 的准确率。所以使用预训练的词向量作为特征是非常有效的。一般来说，在自然语言处理任务中，当样本数量非常少时，使用预训练的词向量是可行的（实际上，预训练的词向量引入了外部语义信息，往往对模型很有帮助）。

以下部分为译者添加

国内的 Rachel-Zhang 用 sklearn 对同样的数据集做过基于传统机器学习算法的实验，请点击这里。 同时 Richard Socher 等在提出 GloVe 词向量的那篇论文中指出 GloVe 词向量比 word2vec 的性能更好[1]。之后的研究表示 word2vec 和 GloVe 其实各有千秋，例如 Schnabel 等提出了用于测评词向量的各项指标，测评显示 word2vec 在大部分测评指标优于 GloVe 和 C&W 词向量[2]。本文实现其实可以利用谷歌新闻的 word2vec 词向量再做一组测评实验。

-
- Getting started
  - Keras FAQ：常见问题 repeated
  - 一些基本概念 repeated
  - Keras 示例程序

addition_rnn.py: 序列到序列学习, 实现两个数的加法

- antirectifier.py: 展示了如何在 Keras 中定制自己的层
- babi_memnn.py: 在 bAbI 数据集上训练一个记忆网络,用于阅读理解
- babi_rnn.py: 在 bAbI 数据集上训练一个循环网络,用于阅读理解
- cifar10_cnn.py: 在 CIFAR10 数据集上训练一个简单的深度 CNN 网络,用于小图片识别
- conv_filter_visualization.py: 通过在输入空间上梯度上升可视化 VGG16 的滤波器

- conv_lstm.py: 展示了一个卷积 LSTM 网络的应用
- deep_dream.py: Google DeepDream 的 Keras 实现
- image_ocr.py:训练了一个卷积+循环网络+CTC logloss 来进行 OCR
- imdb_bidirectional_lstm.py: 在 IMDB 数据集上训练一个双向 LSTM 网络,用于情感分类.
- imdb_cnn.py: 展示了如何在文本分类上如何使用 Covolution1D
- imdb_cnn_lstm.py: 训练了一个栈式的卷积网络+循环网络进行 IMDB 情感分类.
- imdb_fasttext.py: 训练了一个 FastText 模型用于 IMDB 情感分类
- imdb_lstm.py: 训练了一个 LSTM 网络用于 IMDB 情感分类.
- lstm_benchmark.py: 在 IMDB 情感分类上比较了 LSTM 的不同实现的性能
- lstm_text_generation.py: 从尼采的作品中生成文本
- mnist_acgan.py：AC-GAN(Auxiliary Classifier GAN)实现的示例
- mnist_cnn.py: 训练一个用于 mnist 数据集识别的卷积神经网络
- mnist_hierarchical_rnn.py: 训练了一个 HRNN 网络用于 MNIST 数字识别
- mnist_irnn.py: 重现了基于逐像素点序列的 IRNN 实验,文章见 Le et al. "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units"
- mnist_mlp.py: 训练了一个简单的多层感知器用于 MNIST 分类
- mnist_net2net.py: 在 mnist 上重现了文章中的 Net2Net 实验,文章为"Net2Net: Accelerating Learning via Knowledge Transfer".
- mnist_siamese_graph.py:基于 MNIST 训练了一个多层感知器的 Siamese 网络
- mnist_sklearn_wrapper.py: 展示了如何使用 sklearn 包装器
- mnist_swwae.py: 基于残差网络和 MNIST 训练了一个栈式的 What-Where 自动编码器
- mnist_transfer_cnn.py: 迁移学习的小例子
- neural_doodle.py:神经网络绘画
- neural_style_transfer.py: 图像风格转移
- pretrained_word_embeddings.py: 将 GloVe 嵌入层载入固化的 Keras Embedding 层中, 并用以在新闻数据集上训练文本分类模型
- reuters_mlp.py: 训练并评估一个简单的多层感知器进行路透社新闻主题分类
- stateful_lstm.py: 展示了如何使用状态 RNN 对长序列进行建模
- variational_autoencoder.py: 展示了如何搭建变分编码器
- variational_autoencoder_deconv.py Demonstrates how to build a variational autoencoder with Keras using deconvolution layers.

- 快速开始泛型模型

Keras 泛型模型接口是用户定义多输出模型、非循环有向模型或具有共享层的模型等复杂模型的途径
这部分的文档假设你已经对 `Sequential` 模型已经比较熟悉

让我们从简单一点的模型开始

第一个模型：全连接网络

`Sequential` 当然是实现全连接网络的最好方式，但我们从简单的全连接网络开始，有助于我们学习这部分的内容。在开始前，有几个概念需要澄清：

- 层对象接受张量为参数，返回一个张量。张量在数学上只是数据结构的扩充，一阶张量就是向量，二阶张量就是矩阵，三阶张量就是立方体。在这里张量只是广义的表达一种数据结构，例如一张彩色图像其实就是一个三阶张量，它由三个通道的像素值堆叠而成。而 10000 张彩色图构成的一个数据集合则是四阶张量。
- 输入是张量，输出也是张量的一个框架就是一个模型
- 这样的模型可以被像 Keras 的 `Sequential` 一样被训练

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

# this returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# this creates a model that includes
# the Input layer and three Dense layers
model = Model(input=inputs, output=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

所有的模型都是可调用的，就像层一样

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

```
x = Input(shape=(784,))
# this works, and returns the 10-way softmax we defined above.
y = model(x)
```

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

```
from keras.layers import TimeDistributed

# input tensor for sequences of 20 timesteps,
# each containing a 784-dimensional vector
input_sequences = Input(shape=(20, 784))

# this applies our previous model to every timestep in the input sequences.
# the output of the previous model was a 10-way softmax,
# so the output of the layer below will be a sequence of 20 vectors of size 10.
processed_sequences = TimeDistributed(model)(input_sequences)
```

多输入和多输出模型

使用泛型模型的一个典型场景是搭建多输入、多输出的模型。

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

让我们用泛型模型来实现这个框图

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

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

# headline input: meant to receive sequences of 100 integers, between 1 and
10000.
# note that we can name any layer by passing it a "name" argument.
main_input = Input(shape=(100,), dtype='int32', name='main_input')

# this embedding layer will encode the input sequence
# into a sequence of dense 512-dimensional vectors.
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)

# a LSTM will transform the vector sequence into a single vector,
# containing information about the entire sequence
lstm_out = LSTM(32)(x)
```

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

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

再然后，我们将 LSTM 与额外的输入数据串联起来组成输入，送入模型中：

```
auxiliary_input = Input(shape=(5,), name='aux_input')
x = merge([lstm_out, auxiliary_input], mode='concat')

# we stack a deep fully-connected network on top
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)

# and finally we add the main logistic regression layer
main_output = Dense(1, activation='sigmoid', name='main_output')(x)
```

最后，我们定义整个 2 输入，2 输出的模型：

```
model = Model(input=[main_input, auxiliary_input], output=[main_output,
auxiliary_output])
```

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

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

编译完成后，我们通过传递训练数据和目标值训练该模型：

```
model.fit([headline_data, additional_data], [labels, labels],
          nb_epoch=50, batch_size=32)
```

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

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

# and trained it via:
model.fit({'main_input': headline_data, 'aux_input': additional_data},
          {'main_output': labels, 'aux_output': labels},
          nb_epoch=50, batch_size=32)
```

共享层

另一个使用泛型模型的场合是使用共享层的时候。

考虑微博数据，我们希望建立模型来判别两条微博是否是来自同一个用户，这个需求同样可以用来判断一个用户的两条微博的相似性。

一种实现方式是，我们建立一个模型，它分别将两条微博的数据映射到两个特征向量上，然后将特征向量串联并加一个 logistic 回归层，输出它们来自同一个用户的概率。这种模型的训练数据是一对对的微博。

因为这个问题是对称的，所以处理第一条微博的模型当然也能重用于处理第二条微博。所以这里我们使用一个共享的 LSTM 层来进行映射。

首先，我们将微博的数据转为（140，256）的矩阵，即每条微博有 140 个字符，每个单词的特征由一个 256 维的词向量表示，向量的每个元素为 1 表示某个字符出现，为 0 表示不出现，这是一个 one-hot 编码。

【Tips】之所以是（140，256）是因为一条微博最多有 140 个字符（据说现在要取消这个限制了），而扩展的 ASCII 码表编码了常见的 256 个字符。原文中此处为 Tweet，所以对外国人而言这是合理的。如果考虑中文字符，那一个单词的词向量就不止 256 了。【@Bigmoyan】

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

```
from keras.models import Model

tweet_a = Input(shape=(140, 256))
tweet_b = Input(shape=(140, 256))
```

若要对不同的输入共享同一层，就初始化该层一次，然后多次调用它

```
# this layer can take as input a matrix
# and will return a vector of size 64
shared_lstm = LSTM(64)

# when we reuse the same layer instance
# multiple times, the weights of the layer
# are also being reused
# (it is effectively *the same* layer)
encoded_a = shared_lstm(tweet_a)
encoded_b = shared_lstm(tweet_b)

# we can then concatenate the two vectors:
merged_vector = merge([encoded_a, encoded_b], mode='concat', concat_axis=-1)

# and add a logistic regression on top
predictions = Dense(1, activation='sigmoid')(merged_vector)

# we define a trainable model linking the
# tweet inputs to the predictions
model = Model(input=[tweet_a, tweet_b], output=predictions)

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

先暂停一下，看看共享层到底输出了什么，它的输出数据 shape 又是什么

层“节点”的概念

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

在上一版本的 Keras 中，你可以通过 `layer.get_output()`方法来获得层的输出张量，或者通过 `layer.output_shape` 获得其输出张量的 shape。这个版本的 Keras 你仍然可以这么做（除了 `layer.get_output()`被 `output()`替换）。但如果一个层与多个输入相连，会出现什么情况呢?

如果层只与一个输入相连，那没有任何困惑的地方。`.output()`将会返回该层唯一的输出

```
a = Input(shape=(140, 256))

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

assert lstm.output == encoded_a
```

但当层与多个输入相连时，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
```

上面这段代码会报错

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

通过下面这种调用方式即可解决

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

够简单吧?

对于 `input_shape` 和 `output_shape` 也是一样，如果一个层只有一个节点，或所有的节点都有相同的输入或输出 shape，那么 `input_shape` 和 `output_shape` 都是没有歧义的，并也只返回一个值。但是，例如你把一个相同的 `Convolution2D` 应用于一个大小为(3,32,32)的数据，然后又将其应用于一个(3,64,64)的数据，那么此时该层就具有了多个输入和输出的 shape，你就需要显式的指定节点的下标，来表明你想取的是哪个了

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

conv = Convolution2D(16, 3, 3, border_mode='same')
conved_a = conv(a)

# only one input so far, the following will work:
assert conv.input_shape == (None, 3, 32, 32)

conved_b = conv(b)
# now the `.input_shape` property wouldn't work, but this does:
assert conv.get_input_shape_at(0) == (None, 3, 32, 32)
assert conv.get_input_shape_at(1) == (None, 3, 64, 64)
```

更多的例子

代码示例依然是学习的最佳方式，这里是更多的例子

inception 模型

inception 的详细结构参见 Google 的这篇论文：Going Deeper with Convolutions

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

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

```
tower_1 = Convolution2D(64, 1, 1, border_mode='same', activation='relu')
(input_img)
tower_1 = Convolution2D(64, 3, 3, border_mode='same', activation='relu')
(tower_1)

tower_2 = Convolution2D(64, 1, 1, border_mode='same', activation='relu')
(input_img)
tower_2 = Convolution2D(64, 5, 5, border_mode='same', activation='relu')
(tower_2)

tower_3 = MaxPooling2D((3, 3), strides=(1, 1), border_mode='same')(input_img)
tower_3 = Convolution2D(64, 1, 1, border_mode='same', activation='relu')
(tower_3)

output = merge([tower_1, tower_2, tower_3], mode='concat', concat_axis=1)
```

卷积层的残差连接

残差网络（Residual Network）的详细信息请参考这篇文章：Deep Residual Learning for Image Recognition

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

# input tensor for a 3-channel 256x256 image
x = Input(shape=(3, 256, 256))
# 3x3 conv with 3 output channels(same as input channels)
y = Convolution2D(3, 3, 3, border_mode='same')(x)
# this returns x + y.
z = merge([x, y], mode='sum')
```

共享视觉模型

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

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

# first, define the vision modules
digit_input = Input(shape=(1, 27, 27))
x = Convolution2D(64, 3, 3)(digit_input)
x = Convolution2D(64, 3, 3)(x)
x = MaxPooling2D((2, 2))(x)
out = Flatten()(x)

vision_model = Model(digit_input, out)

# then define the tell-digits-apart model
digit_a = Input(shape=(1, 27, 27))
digit_b = Input(shape=(1, 27, 27))

# the vision model will be shared, weights and all
out_a = vision_model(digit_a)
out_b = vision_model(digit_b)

concatenated = merge([out_a, out_b], mode='concat')
out = Dense(1, activation='sigmoid')(concatenated)

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

视觉问答模型

在针对一幅图片使用自然语言进行提问时，该模型能够提供关于该图片的一个单词的答案

这个模型将自然语言的问题和图片分别映射为特征向量，将二者合并后训练一个 logistic 回归层，从一系列可能的回答中挑选一个。

```
from keras.layers import Convolution2D, MaxPooling2D, Flatten
from keras.layers import Input, LSTM, Embedding, Dense, merge
from keras.models import Model, Sequential

# first, let's define a vision model using a Sequential model.
# this model will encode an image into a vector.
vision_model = Sequential()
vision_model.add(Convolution2D(64, 3, 3, activation='relu', border_mode='same',
input_shape=(3, 224, 224)))
vision_model.add(Convolution2D(64, 3, 3, activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Convolution2D(128, 3, 3, activation='relu',
border_mode='same'))
vision_model.add(Convolution2D(128, 3, 3, activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Convolution2D(256, 3, 3, activation='relu',
border_mode='same'))
vision_model.add(Convolution2D(256, 3, 3, activation='relu'))
vision_model.add(Convolution2D(256, 3, 3, activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Flatten())

# now let's get a tensor with the output of our vision model:
image_input = Input(shape=(3, 224, 224))
encoded_image = vision_model(image_input)

# next, let's define a language model to encode the question into a vector.
# each question will be at most 100 word long,
# and we will index words as integers from 1 to 9999.
question_input = Input(shape=(100,), dtype='int32')
embedded_question = Embedding(input_dim=10000, output_dim=256, input_length=100)
(question_input)
encoded_question = LSTM(256)(embedded_question)

# let's concatenate the question vector and the image vector:
merged = merge([encoded_question, encoded_image], mode='concat')

# and let's train a logistic regression over 1000 words on top:
output = Dense(1000, activation='softmax')(merged)

# this is our final model:
vqa_model = Model(input=[image_input, question_input], output=output)

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

视频问答模型

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

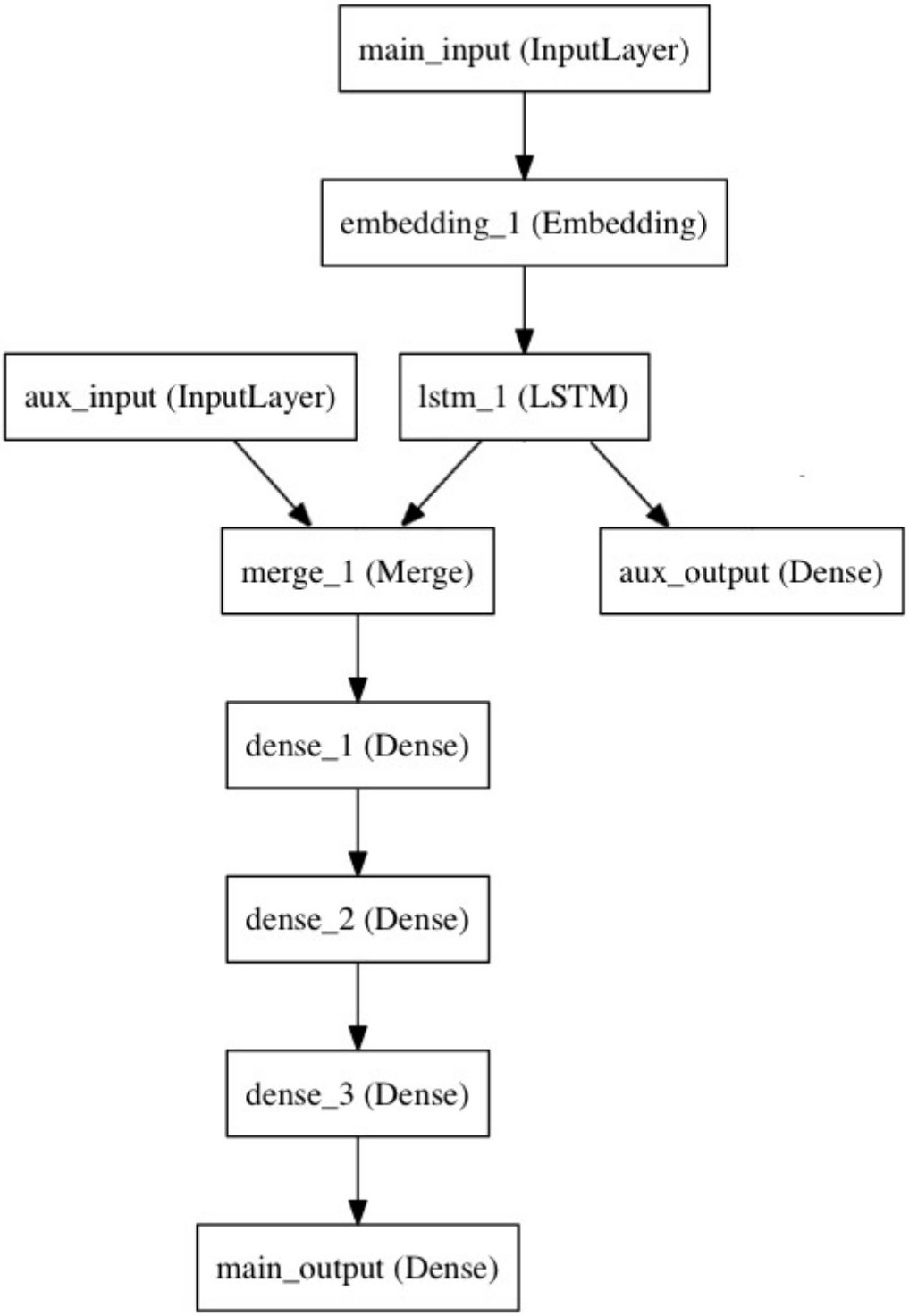
```
from keras.layers import TimeDistributed
```

```
video_input = Input(shape=(100, 3, 224, 224))
# this is our video encoded via the previously trained vision_model (weights are
reused)
encoded_frame_sequence = TimeDistributed(vision_model)(video_input)  # the
output will be a sequence of vectors
encoded_video = LSTM(256)(encoded_frame_sequence)  # the output will be a vector

# this is a model-level representation of the question encoder, reusing the same
weights as before:
question_encoder = Model(input=question_input, output=encoded_question)

# let's use it to encode the question:
video_question_input = Input(shape=(100,), dtype='int32')
encoded_video_question = question_encoder(video_question_input)

# and this is our video question answering model:
merged = merge([encoded_video, encoded_video_question], mode='concat')
output = Dense(1000, activation='softmax')(merged)
video_qa_model = Model(input=[video_input, video_question_input], output=output)
```



    - 声明 repeated
    - 声明 repeated
    - Sequential model repeated
    - Keras 使用陷阱 repeated
- Layers
    - 关于 Keras 的“层”（Layer） repeated
    - 高级激活层 Advanced Activation repeated
    - 卷积层 repeated
    - 常用层 repeated

        - 嵌入层 Embedding repeated
        - 局部连接层 LocallyConnceted repeated
        - 噪声层 Noise repeated
        - （批）规范化 BatchNormalization repeated
        - 池化层 repeated
        - 递归层 Recurrent repeated
        - 包装器 Wrapper repeated
        - Writting layer repeated
    - Models
        - 关于 Keras 模型

Keras 有两种类型的模型，顺序模型（Sequential）和泛型模型（Model）

两类模型有一些方法是相同的：

- `model.summary()`：打印出模型概况
- `model.get_config()`:返回包含模型配置信息的 Python 字典。模型也可以从它的 config 信息中重构回去

```
config = model.get_config()
model = Model.from_config(config)
# or, for Sequential
model = Sequential.from_config(config)
```

- `model.get_weights()`：返回模型权重张量的列表，类型为 numpy array
- `model.set_weights()`：从 numpy array 里将权重载入给模型，要求数组具有与 `model.get_weights()`相同的形状。
- `model.to_json`：返回代表模型的 JSON 字符串，仅包含网络结构，不包含权值。可以从 JSON 字符串中重构原模型：

```
from models import model_from_json

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

- `model.to_yaml`：与 `model.to_json` 类似，同样可以从产生的 YAML 字符串中重构模型

```
from models import model_from_yaml

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

- `model.save_weights(filepath)`：将模型权重保存到指定路径，文件类型是 HDF5（后缀是.h5）
- `model.load_weights(filepath, by_name=False)`：从 HDF5 文件中加载权重到当前模型中, 默认情况下模型的结构将保持不变。如果想将权重载入不同的模型（有些层相同）中，则设置 `by_name=True`，只有名字匹配的层才会载入权重

- 泛型模型接口

为什么叫“泛型模型”，请查看一些基本概念

Keras 的泛型模型为 `Model`，即广义的拥有输入和输出的模型，我们使用 `Model` 来初始化一个泛型模型

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

a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(input=a, output=b)
```

在这里，我们的模型以 a 为输入，以 b 为输出，同样我们可以构造拥有多输入和多输出的模型

```
model = Model(input=[a1, a2], output=[b1, b3, b3])
```

常用 Model 属性

- `model.layers`：组成模型图的各个层
- `model.inputs`：模型的输入张量列表
- `model.outputs`：模型的输出张量列表

Model 模型方法

compile

```
compile(self, optimizer, loss, metrics=[], loss_weights=None,
sample_weight_mode=None)
```

本函数编译模型以供训练，参数有

- optimizer：优化器，为预定义优化器名或优化器对象，参考优化器
- loss：目标函数，为预定义损失函数名或一个目标函数，参考目标函数
- metrics：列表，包含评估模型在训练和测试时的性能的指标，典型用法是 `metrics=['accuracy']`如果要在多输出模型中为不同的输出指定不同的指标，可像该参数传递一个字典，例如 `metrics={'ouput_a': 'accuracy'}`
- sample_weight_mode：如果你需要按时间步为样本赋权（2D 权矩阵），将该值设为“temporal”。默认为“None”，代表按样本赋权（1D 权）。如果模型有多个输出，可以向该参数传入指定 sample_weight_mode 的字典或列表。在下面 `fit` 函数的解释中有相关的参考内容。
- kwargs：使用 TensorFlow 作为后端请忽略该参数，若使用 Theano 作为后端，kwargs 的值将会传递给 K.function

【Tips】如果你只是载入模型并利用其 predict，可以不用进行 compile。在 Keras 中，compile 主要完成损失函数和优化器的一些配置，是为训练服务的。predict 会在内部进行符号函数的编译工作（通过调用_make_predict_function 生成函数）【@白菜，@我是小将】

---

fit

```
fit(self, x, y, batch_size=32, nb_epoch=10, verbose=1, callbacks=[],
validation_split=0.0, validation_data=None, shuffle=True, class_weight=None,
sample_weight=None)
```

本函数用以训练模型，参数有：

- x：输入数据。如果模型只有一个输入，那么 x 的类型是 numpy array，如果模型有多个输入，那么 x 的类型应当为 list，list 的元素是对应于各个输入的 numpy array。如果模型的每个输入都有名字，则可以传入一个字典，将输入名与其输入数据对应起来。
- y：标签，numpy array。如果模型有多个输出，可以传入一个 numpy array 的 list。如果模型的输出拥有名字，则可以传入一个字典，将输出名与其标签对应起来。
- batch_size：整数，指定进行梯度下降时每个 batch 包含的样本数。训练时一个 batch 的样本会被计算一次梯度下降，使目标函数优化一步。
- nb_epoch：整数，训练的轮数，训练数据将会被遍历 nb_epoch 次。Keras 中 nb 开头的变量均为"number of"的意思
- verbose：日志显示，0 为不在标准输出流输出日志信息，1 为输出进度条记录，2 为每个 epoch 输出一行记录
- callbacks：list，其中的元素是 `keras.callbacks.Callback` 的对象。这个 list 中的回调函数将会在训练过程中的适当时机被调用，参考回调函数
- validation_split：0~1 之间的浮点数，用来指定训练集的一定比例数据作为验证集。验证集将不参与训练，并在每个 epoch 结束后测试的模型的指标，如损失函数、精确度等。
- validation_data：形式为（X，y）或（X，y，sample_weights）的 tuple，是指定的验证集。此参数将覆盖 validation_spilt。
- shuffle：布尔值，表示是否在训练过程中每个 epoch 前随机打乱输入样本的顺序。
- class_weight：字典，将不同的类别映射为不同的权值，该参数用来在训练过程中调整损失函数（只能用于训练）。该参数在处理非平衡的训练数据（某些类的训练样本数很少）时，可以使得损失函数对样本数不足的数据更加关注。
- sample_weight：权值的 numpy array，用于在训练时调整损失函数（仅用于训练）。可以传递一个 1D 的与样本等长的向量用于对样本进行 1 对 1 的加权，或者在面对时序数据时，传递一个的形式为（samples，sequence_length）的矩阵来为每个时间步上的样本赋不同的权。这种情况下请确定在编译模型时添加了 `sample_weight_mode='temporal'`。

`fit` 函数返回一个 `History` 的对象，其 `History.history` 属性记录了损失函数和其他指标的数值随 epoch 变化的情况，如果有验证集的话，也包含了验证集的这些指标变化情况

---

evaluate

```
evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=None)
```

本函数按 batch 计算在某些输入数据上模型的误差，其参数有：

- x：输入数据，与 `fit` 一样，是 numpy array 或 numpy array 的 list
- y：标签，numpy array
- batch_size：整数，含义同 `fit` 的同名参数
- verbose：含义同 `fit` 的同名参数，但只能取 0 或 1
- sample_weight：numpy array，含义同 `fit` 的同名参数

本函数返回一个测试误差的标量值（如果模型没有其他评价指标），或一个标量的 list（如果模型还有其他的评价指标）。`model.metrics_names` 将给出 list 中各个值的含义。

如果没有特殊说明，以下函数的参数均保持与 `fit` 的同名参数相同的含义

如果没有特殊说明，以下函数的 verbose 参数（如果有）均只能取 0 或 1

---

predict

```
predict(self, x, batch_size=32, verbose=0)
```

本函数按 batch 获得输入数据对应的输出，其参数有：

函数的返回值是预测值的 numpy array

---

train_on_batch

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

本函数在一个 batch 的数据上进行一次参数更新

函数返回训练误差的标量值或标量值的 list，与 evaluate 的情形相同。

---

test_on_batch

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

本函数在一个 batch 的样本上对模型进行评估

函数的返回与 evaluate 的情形相同

---

predict_on_batch

```
predict_on_batch(self, x)
```

本函数在一个 batch 的样本上对模型进行测试

函数返回模型在一个 batch 上的预测结果

---

fit_generator

```
fit_generator(self, generator, samples_per_epoch, nb_epoch, verbose=1,
callbacks=[], validation_data=None, nb_val_samples=None, class_weight={},
max_q_size=10)
```

利用 Python 的生成器，逐个生成数据的 batch 并进行训练。生成器与模型将并行执行以提高效率。例如，该函数允许我们在 CPU 上进行实时的数据提升，同时在 GPU 上进行模型训练

函数的参数是：

- generator：生成器函数，生成器的输出应该为：
    - 一个形如（inputs，targets）的 tuple
    - 一个形如（inputs, targets,sample_weight）的 tuple。所有的返回值都应该包含相同数目的样本。生成器将无限在数据集上循环。每个 epoch 以经过模型的样本数达到 `samples_per_epoch` 时，记一个 epoch 结束
- samples_per_epoch：整数，当模型处理的样本达到此数目时计一个 epoch 结束，执行下一个 epoch
- verbose：日志显示，0 为不在标准输出流输出日志信息，1 为输出进度条记录，2 为每个 epoch 输出一行记录
- validation_data：具有以下三种形式之一
    - 生成验证集的生成器
    - 一个形如（inputs,targets）的 tuple
    - 一个形如（inputs,targets，sample_weights）的 tuple
- nb_val_samples：仅当 `validation_data` 是生成器时使用，用以限制在每个 epoch 结束时用来验证模型的验证集样本数，功能类似于 `samples_per_epoch`
- max_q_size：生成器队列的最大容量

函数返回一个 `History` 对象

例子

```
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'),
        samples_per_epoch=10000, nb_epoch=10)
```

---

evaluate_generator

`evaluate_generator(self, generator, val_samples, max_q_size=10)`

本函数使用一个生成器作为数据源，来评估模型，生成器应返回与 `test_on_batch` 的输入数据相同类型的数据。

函数的参数是：

- generator：生成输入 batch 数据的生成器
- val_samples：生成器应该返回的总样本数
- max_q_size：生成器队列的最大容量
- nb_worker：使用基于进程的多线程处理时的进程数
- pickle_safe：若设置为 True，则使用基于进程的线程。注意因为它的实现依赖于多进程处理，不可传递不可 pickle 的参数到生成器中，因为它们不能轻易的传递到子进程中。

---

predict_generator

`predict_generator(self, generator, val_samples, max_q_size=10, nb_worker=1, pickle_safe=False)`

从一个生成器上获取数据并进行预测，生成器应返回与 `predict_on_batch` 输入类似的数据

函数的参数是：

- generator：生成输入 batch 数据的生成器
- val_samples：生成器应该返回的总样本数
- max_q_size：生成器队列的最大容量
- nb_worker：使用基于进程的多线程处理时的进程数
- pickle_safe：若设置为 True，则使用基于进程的线程。注意因为它的实现依赖于多进程处理，不可传递不可 pickle 的参数到生成器中，因为它们不能轻易的传递到子进程中。

---

get_layer

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

本函数依据模型中层的下标或名字获得层对象，泛型模型中层的下标依据自底向上，水平遍历的顺序。

- name：字符串，层的名字
- index： 整数，层的下标

函数的返回值是层对象

- Sequential 模型接口

如果刚开始学习 Sequential 模型，请首先移步这里阅读文档

常用 Sequential 属性

- `model.layers` 是添加到模型上的层的 list

Sequential 模型方法

compile

```
compile(self, optimizer, loss, metrics=[], sample_weight_mode=None)
```

编译用来配置模型的学习过程，其参数有

- optimizer：字符串（预定义优化器名）或优化器对象，参考优化器
- loss：字符串（预定义损失函数名）或目标函数，参考目标函数
- metrics：列表，包含评估模型在训练和测试时的网络性能的指标，典型用法是 `metrics=['accuracy']`
- sample_weight_mode：如果你需要按时间步为样本赋权（2D 权矩阵），将该值设为“temporal”。默认为“None”，代表按样本赋权（1D 权）。在下面 `fit` 函数的解释中有相关的参考内容。
- kwargs：使用 TensorFlow 作为后端请忽略该参数，若使用 Theano 作为后端，kwargs 的值将会传递给 K.function

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

fit

```
fit(self, x, y, batch_size=32, nb_epoch=10, verbose=1, callbacks=[],
validation_split=0.0, validation_data=None, shuffle=True, class_weight=None,
sample_weight=None)
```

本函数将模型训练 nb_epoch 轮，其参数有：

- x：输入数据。如果模型只有一个输入，那么 x 的类型是 numpy array，如果模型有多个输入，那么 x 的类型应当为 list，list 的元素是对应于各个输入的 numpy array
- y：标签，numpy array
- batch_size：整数，指定进行梯度下降时每个 batch 包含的样本数。训练时一个 batch 的样本会被计算一次梯度下降，使目标函数优化一步。
- nb_epoch：整数，训练的轮数，训练数据将会被遍历 nb_epoch 次。Keras 中 nb 开头的变量均为"number of"的意思
- verbose：日志显示，0 为不在标准输出流输出日志信息，1 为输出进度条记录，2 为每个 epoch 输出一行记录
- callbacks：list，其中的元素是 `keras.callbacks.Callback` 的对象。这个 list 中的回调函数将会在训练过程中的适当时机被调用，参考回调函数
- validation_split：0~1 之间的浮点数，用来指定训练集的一定比例数据作为验证集。验证集将不参与训练，并在每个 epoch 结束后测试的模型的指标，如损失函数、精确度等。
- validation_data：形式为（X，y）的 tuple，是指定的验证集。此参数将覆盖 validation_spilt。
- shuffle：布尔值或字符串，一般为布尔值，表示是否在训练过程中随机打乱输入样本的顺序。若为字符串“batch”，则是用来处理 HDF5 数据的特殊情况，它将在 batch 内部将数据打乱。
- class_weight：字典，将不同的类别映射为不同的权值，该参数用来在训练过程中调整损失函数（只能用于训练）
- sample_weight：权值的 numpy array，用于在训练时调整损失函数（仅用于训练）。可以传递一个 1D 的与样本等长的向量用于对样本进行 1 对 1 的加权，或者在面对时序数据时，传递一个的形式为（samples，sequence_length）的矩阵来为每个时间步上的样本赋不同的权。这种情况下请确定在编译模型时添加了 `sample_weight_mode='temporal'`。

`fit` 函数返回一个 `History` 的对象，其 `History.history` 属性记录了损失函数和其他指标的数值随 epoch 变化的情况，如果有验证集的话，也包含了验证集的这些指标变化情况

---

evaluate

```
evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=None)
```

本函数按 batch 计算在某些输入数据上模型的误差，其参数有：

- x：输入数据，与 `fit` 一样，是 numpy array 或 numpy array 的 list

- y：标签，numpy array
- batch_size：整数，含义同 `fit` 的同名参数
- verbose：含义同 `fit` 的同名参数，但只能取 0 或 1
- sample_weight：numpy array，含义同 `fit` 的同名参数

本函数返回一个测试误差的标量值（如果模型没有其他评价指标），或一个标量的 list（如果模型还有其他的评价指标）。`model.metrics_names` 将给出 list 中各个值的含义。

如果没有特殊说明，以下函数的参数均保持与 `fit` 的同名参数相同的含义

如果没有特殊说明，以下函数的 verbose 参数（如果有）均只能取 0 或 1

---

predict

```
predict(self, x, batch_size=32, verbose=0)
```

本函数按 batch 获得输入数据对应的输出，其参数有：

函数的返回值是预测值的 numpy array

---

predict_classes

```
predict_classes(self, x, batch_size=32, verbose=1)
```

本函数按 batch 产生输入数据的类别预测结果

函数的返回值是类别预测结果的 numpy array 或 numpy

---

predict_proba

```
predict_proba(self, x, batch_size=32, verbose=1)
```

本函数按 batch 产生输入数据属于各个类别的概率

函数的返回值是类别概率的 numpy array

---

train_on_batch

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

本函数在一个 batch 的数据上进行一次参数更新

函数返回训练误差的标量值或标量值的 list，与 evaluate 的情形相同。

---

test_on_batch

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

本函数在一个 batch 的样本上对模型进行评估

函数的返回与 evaluate 的情形相同

---

predict_on_batch

```
predict_on_batch(self, x)
```

本函数在一个 batch 的样本上对模型进行测试

函数返回模型在一个 batch 上的预测结果

---

fit_generator

```
fit_generator(self, generator, samples_per_epoch, nb_epoch, verbose=1,
callbacks=[], validation_data=None, nb_val_samples=None, class_weight=None,
max_q_size=10)
```

利用 Python 的生成器，逐个生成数据的 batch 并进行训练。生成器与模型将并行执行以提高效率。例如，该函数允许我们在 CPU 上进行实时的数据提升，同时在 GPU 上进行模型训练

函数的参数是：

- generator：生成器函数，生成器的输出应该为：
    - 一个形如（inputs，targets）的 tuple
    - 一个形如（inputs, targets,sample_weight）的 tuple。所有的返回值都应该包含相同数目的样本。生成器将无限在数据集上循环。每个 epoch 以经过模型的样本数达到 `samples_per_epoch` 时，记一个 epoch 结束
- samples_per_epoch：整数，当模型处理的样本达到此数目时计一个 epoch 结束，执行下一个 epoch
- verbose：日志显示，0 为不在标准输出流输出日志信息，1 为输出进度条记录，2 为每个 epoch 输出一行记录
- validation_data：具有以下三种形式之一
    - 生成验证集的生成器
    - 一个形如（inputs,targets）的 tuple
    - 一个形如（inputs,targets，sample_weights）的 tuple
- nb_val_samples：仅当 `validation_data` 是生成器时使用，用以限制在每个 epoch 结束时用来验证模型的验证集样本数，功能类似于 `samples_per_epoch`

- max_q_size：生成器队列的最大容量

函数返回一个 `History` 对象

例子：

```
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'),
        samples_per_epoch=10000, nb_epoch=10)
```

---

evaluate_generator

```
evaluate_generator(self, generator, val_samples, max_q_size=10)
```

本函数使用一个生成器作为数据源评估模型，生成器应返回与 `test_on_batch` 的输入数据相同类型的数据。该函数的参数与 `fit_generator` 同名参数含义相同

- Other
  - 激活函数 Activations

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.core import Activation, Dense

model.add(Dense(64))
model.add(Activation('tanh'))
```

等价于

```
model.add(Dense(64, activation='tanh'))
```

也可以通过传递一个逐元素运算的 Theano/TensorFlow 函数来作为激活函数：

```
from keras import backend as K

def tanh(x):
    return K.tanh(x)

model.add(Dense(64, activation=tanh))
model.add(Activation(tanh)
```

预定义激活函数

- softmax：对输入数据的最后一维进行 softmax，输入数据应形如`(nb_samples, nb_timesteps, nb_dims)`或`(nb_samples,nb_dims)`
- softplus
- softsign
- relu
- tanh
- sigmoid
- hard_sigmoid
- linear

高级激活函数

对于简单的 Theano/TensorFlow 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

- Application 应用

Kera 的应用模块 Application 提供了带有预训练权重的 Keras 模型，这些模型可以用来进行预测、特征提取和 finetune
模型的预训练权重将下载到`~/.keras/models/`并在载入模型时自动载入

可用的模型

应用于图像分类的模型,权重训练自 ImageNet： *Xception* VGG16 *VGG19* ResNet50 * InceptionV3

所有的这些模型(除了 Xception)都兼容 Theano 和 Tensorflow，并会自动基于`~/.keras/keras.json` 的 Keras 的图像维度进行自动设置。例如，如果你设置`image_dim_ordering=tf`，则加载的模型将按照 TensorFlow 的维度顺序来构造，即“Width-Height-Depth”的顺序

应用于音乐自动标签(以 Mel-spectrograms 为输入)

- MusicTaggerCRNN

图片分类模型的示例

利用 ResNet50 网络进行 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', 0.1122357), (u'n02504458', u'African_elephant', 0.061040461)]
```

利用 VGG16 提取特征

```
from keras.applications.vgg16 import VGG16
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
import numpy as np

model = VGG16(weights='imagenet', include_top=False)

img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

features = model.predict(x)
```

从 VGG19 的任意中间层中抽取特征

```
from keras.applications.vgg19 import VGG19
from keras.preprocessing import image
from keras.applications.vgg19 import preprocess_input
from keras.models import Model
import numpy as np

base_model = VGG19(weights='imagenet')
model = Model(input=base_model.input,
output=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)
```

利用新数据集 finetune 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

# create the base pre-trained model
base_model = InceptionV3(weights='imagenet', include_top=False)

# add a global spatial average pooling layer
x = base_model.output
x = GlobalAveragePooling2D()(x)
# let's add a fully-connected layer
x = Dense(1024, activation='relu')(x)
# and a logistic layer -- let's say we have 200 classes
predictions = Dense(200, activation='softmax')(x)

# this is the model we will train
model = Model(input=base_model.input, output=predictions)

# first: train only the top layers (which were randomly initialized)
# i.e. freeze all convolutional InceptionV3 layers
for layer in base_model.layers:
    layer.trainable = False

# compile the model (should be done *after* setting layers to non-trainable)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

# train the model on the new data for a few epochs
model.fit_generator(...)

# at this point, the top layers are well trained and we can start fine-tuning
# convolutional layers from inception V3. We will freeze the bottom N layers
# and train the remaining top layers.

# let's visualize layer names and layer indices to see how many layers
# we should freeze:
for i, layer in enumerate(base_model.layers):
   print(i, layer.name)

# we chose to train the top 2 inception blocks, i.e. we will freeze
# the first 172 layers and unfreeze the rest:
for layer in model.layers[:172]:
   layer.trainable = False
for layer in model.layers[172:]:
   layer.trainable = True

# we need to recompile the model for these modifications to take effect
# we use SGD with a low learning rate
from keras.optimizers import SGD
model.compile(optimizer=SGD(lr=0.0001, momentum=0.9),
loss='categorical_crossentropy')

# we train our model again (this time fine-tuning the top 2 inception blocks
# alongside the top Dense layers
model.fit_generator(...)
```

在定制的输入 tensor 上构建 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_dim_ordering()
== 'tf'

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

模型文档

- Xception
- VGG16
- VGG19
- ResNet50
- InceptionV3
- MusicTaggerCRNN

Xception 模型

```
keras.applications.xception.Xception(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, classes=1000)
```

Xception V1 模型, 权重由 ImageNet 训练而言

在 ImageNet 上,该模型取得了验证集 top1 0.790 和 top5 0.945 的正确率

注意,该模型目前仅能以 TensorFlow 为后端使用,由于它依赖于"SeparableConvolution"层,目前该模型只支持 tf 的维度顺序(width, height, channels)

默认输入图片大小为 299x299

参数

- include_top：是否保留顶层的 3 个全连接网络
- weights：None 代表随机初始化，即不加载预训练权重。'imagenet'代表加载预训练权重
- input_tensor：可填入 Keras tensor 作为模型的图像输出 tensor
- input_shape：可选，仅当 `include_top=False` 有效，应为长为 3 的 tuple，指明输入图片的 shape，图片的宽高必须大于 71，如(150,150,3)
- classes：可选，图片分类的类别数，仅当 `include_top=True` 并且不加载预训练权重时可用。

返回值

Keras 模型对象

参考文献

- Xception: Deep Learning with Depthwise Separable Convolutions

License

预训练权重由我们自己训练而来，基于 MIT license 发布

VGG16 模型

```
keras.applications.vgg16.VGG16(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, classes=1000)
```

VGG16 模型,权重由 ImageNet 训练而来

该模型再 Theano 和 TensorFlow 后端均可使用,并接受 th 和 tf 两种输入维度顺序

模型的默认输入尺寸时 224x224

参数

- include_top：是否保留顶层的 3 个全连接网络
- weights：None 代表随机初始化，即不加载预训练权重。'imagenet'代表加载预训练权重
- input_tensor：可填入 Keras tensor 作为模型的图像输出 tensor
- input_shape：可选，仅当 `include_top=False` 有效，应为长为 3 的 tuple，指明输入图片的 shape，图片的宽高必须大于 48，如(200,200,3)

返回值

- classes：可选，图片分类的类别数，仅当 `include_top=True` 并且不加载预训练权重时可用。

Keras 模型对象

参考文献

- Very Deep Convolutional Networks for Large-Scale Image Recognition：如果在研究中使用了 VGG，请引用该文

License

预训练权重由牛津 VGG 组发布的预训练权重移植而来，基于 Creative Commons Attribution License

---

VGG19 模型

```
keras.applications.vgg19.VGG19(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, classes=1000)
```

VGG19 模型,权重由 ImageNet 训练而来

该模型再 Theano 和 TensorFlow 后端均可使用,并接受 th 和 tf 两种输入维度顺序

模型的默认输入尺寸时 224x224

参数

- include_top：是否保留顶层的 3 个全连接网络
- weights：None 代表随机初始化，即不加载预训练权重。'imagenet'代表加载预训练权重
- input_tensor：可填入 Keras tensor 作为模型的图像输出 tensor
- input_shape：可选，仅当 `include_top=False` 有效，应为长为 3 的 tuple，指明输入图片的 shape，图片的宽高必须大于 48，如(200,200,3)
- classes：可选，图片分类的类别数，仅当 `include_top=True` 并且不加载预训练权重时可用。

返回值

返回值

Keras 模型对象

参考文献

- Very Deep Convolutional Networks for Large-Scale Image Recognition：如果在研究中使用了 VGG，请引用该文

License

预训练权重由牛津 VGG 组发布的预训练权重移植而来，基于 Creative Commons Attribution License

---

ResNet50 模型

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

50 层残差网络模型,权重训练自 ImageNet

该模型再 Theano 和 TensorFlow 后端均可使用,并接受 th 和 tf 两种输入维度顺序

模型的默认输入尺寸时 224x224

参数

- include_top：是否保留顶层的全连接网络
- weights：None 代表随机初始化，即不加载预训练权重。'imagenet'代表加载预训练权重
- input_tensor：可填入 Keras tensor 作为模型的图像输出 tensor
- input_shape：可选，仅当 `include_top=False` 有效，应为长为 3 的 tuple，指明输入图片的 shape，图片的宽高必须大于 197，如(200,200,3)

- classes：可选，图片分类的类别数，仅当 `include_top=True` 并且不加载预训练权重时可用。

返回值

Keras 模型对象

参考文献

- Deep Residual Learning for Image Recognition：如果在研究中使用了 ResNet50，请引用该文

License

预训练权重由 Kaiming He 发布的预训练权重移植而来，基于 MIT License

InceptionV3 模型

```
keras.applications.inception_v3.InceptionV3(include_top=True,
weights='imagenet', input_tensor=None, input_shape=None, classes=1000)
```

InceptionV3 网络,权重训练自 ImageNet

该模型再 Theano 和 TensorFlow 后端均可使用,并接受 th 和 tf 两种输入维度顺序

模型的默认输入尺寸时 299x299

参数

- include_top：是否保留顶层的全连接网络
- weights：None 代表随机初始化，即不加载预训练权重。'imagenet'代表加载预训练权重
- input_tensor：可填入 Keras tensor 作为模型的图像输出 tensor
- classes：可选，图片分类的类别数，仅当 `include_top=True` 并且不加载预训练权重时可用。

返回值

Keras 模型对象

参考文献

- Rethinking the Inception Architecture for Computer Vision：如果在研究中使用了 InceptionV3，请引用该文

License

预训练权重由我们自己训练而来，基于 MIT License

MusicTaggerCRNN 模型

```
keras.applications.music_tagger_crnn.MusicTaggerCRNN(weights='msd',
input_tensor=None, include_top=True, classes=50)
```

该模型时一个卷积循环模型,以向量化的 MelSpectrogram 音乐数据为输入,能够输出音乐的风格. 你可以用 `keras.applications.music_tagger_crnn.preprocess_input` 来将一个音乐文件向量化为 spectrogram.注意,使用该功能需要安装 Librosa,请参考下面的使用范例.

参数

- include_top：是否保留顶层的 1 层全连接网络,若设置为 False,则网络输出 32 维的特征
- weights：None 代表随机初始化，即不加载预训练权重。'msd'代表加载预训练权重(训练自 Million Song Dataset)
- input_tensor：可填入 Keras tensor 作为模型的输出 tensor,如使用 layer.input 选用一层的输入张量为模型的输入张量.

返回值

Keras 模型对象

参考文献

- Convolutional Recurrent Neural Networks for Music Classification

License

预训练权重由我们自己训练而来，基于 MIT License

使用范例:音乐特征抽取与风格标定

```
from keras.applications.music_tagger_crnn import MusicTaggerCRNN
from keras.applications.music_tagger_crnn import preprocess_input,
decode_predictions
import numpy as np

# 1. Tagging
model = MusicTaggerCRNN(weights='msd')

audio_path = 'audio_file.mp3'
melgram = preprocess_input(audio_path)
melgrams = np.expand_dims(melgram, axis=0)

preds = model.predict(melgrams)
print('Predicted:')
print(decode_predictions(preds))
# print: ('Predicted:', [[('rock', 0.097071797), ('pop', 0.042456303),
('alternative', 0.032439161), ('indie', 0.024491295), ('female vocalists',
0.016455274)]])

#. 2. Feature extraction
model = MusicTaggerCRNN(weights='msd', include_top=False)

audio_path = 'audio_file.mp3'
```

```
melgram = preprocess_input(audio_path)
melgrams = np.expand_dims(melgram, axis=0)

feats = model.predict(melgrams)
print('Features:')
print(feats[0, :10])
# print: ('Features:', [-0.19160545 0.94259131 -0.9991011 0.47644514 -0.19089699
0.99033844 0.1103896 -0.00340496 0.14823607 0.59856361])
```

- 回调函数 Callbacks

回调函数是一组在训练的特定阶段被调用的函数集，你可以使用回调函数来观察训练过程中网络内部的状态和统计信息。通过传递回调函数列表到模型的`.fit()`中，即可在给定的训练阶段调用该函数集中的函数。

【Tips】虽然我们称之为回调“函数”，但事实上 Keras 的回调函数是一个类，回调函数只是习惯性称呼

CallbackList

`keras.callbacks.CallbackList(callbacks=[], queue_length=10)`

Callback

`keras.callbacks.Callback()`

这是回调函数的抽象类，定义新的回调函数必须继承自该类

类属性

- params：字典，训练参数集（如信息显示方法 verbosity，batch 大小，epoch 数）
- model：`keras.models.Model` 对象，为正在训练的模型的引用

回调函数以字典 `logs` 为参数，该字典包含了一系列与当前 batch 或 epoch 相关的信息。

目前，模型的`.fit()`中有下列参数会被记录到 `logs` 中：

- 在每个 epoch 的结尾处（on_epoch_end），`logs` 将包含训练的正确率和误差，`acc` 和 `loss`，如果指定了验证集，还会包含验证集正确率和误差 `val_acc)`和 `val_loss`，`val_acc` 还额外需要在`.compile` 中启用 `metrics=['accuracy']`。
- 在每个 batch 的开始处（on_batch_begin）：`logs` 包含 `size`，即当前 batch 的样本数
- 在每个 batch 的结尾处（on_batch_end）：`logs` 包含 `loss`，若启用 `accuracy` 则还包含 `acc`

BaseLogger

`keras.callbacks.BaseLogger()`

该回调函数用来对每个 epoch 累加 `metrics` 指定的监视指标的 epoch 平均值

该回调函数在每个 Keras 模型中都会被自动调用

ProgbarLogger

```
keras.callbacks.ProgbarLogger()
```

该回调函数用来将 `metrics` 指定的监视指标输出到标准输出上

History

```
keras.callbacks.History()
```

该回调函数在 Keras 模型上会被自动调用，`History` 对象即为 `fit` 方法的返回值

ModelCheckpoint

```
keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0,
save_best_only=False, save_weights_only=False, mode='auto', period=1)
```

该回调函数将在每个 epoch 后保存模型到 `filepath`

`filepath` 可以是格式化的字符串，里面的占位符将会被 `epoch` 值和传入 `on_epoch_end` 的 `logs` 关键字所填入

例如，`filepath` 若为 `weights.{epoch:02d-{val_loss:.2f}}.hdf5`，则会生成对应 epoch 和验证集 loss 的多个文件。

参数

- filename：字符串，保存模型的路径
- monitor：需要监视的值
- verbose：信息展示模式，0 或 1
- save_best_only：当设置为 `True` 时，将只保存在验证集上性能最好的模型
- mode：‘auto’，‘min’，‘max’之一，在 `save_best_only=True` 时决定性能最佳模型的评判准则，例如，当监测值为 `val_acc` 时，模式应为 `max`，当检测值为 `val_loss` 时，模式应为 `min`。在 `auto` 模式下，评价准则由被监测值的名字自动推断。
- save_weights_only：若设置为 True，则只保存模型权重，否则将保存整个模型（包括模型结构，配置信息等）
- period：CheckPoint 之间的间隔的 epoch 数

EarlyStopping

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

当监测值不再改善时，该回调函数将中止训练

参数

- monitor：需要监视的量
- patience：当 early stop 被激活（如发现 loss 相比上一个 epoch 训练没有下降），则经过 `patience` 个 epoch 后停止训练。
- verbose：信息展示模式
- mode：‘auto’，‘min’，‘max’之一，在 `min` 模式下，如果检测值停止下降则中止训练。在 `max` 模式下，当检测值不再上升则停止训练。

RemoteMonitor

```
keras.callbacks.RemoteMonitor(root='http://localhost:9000')
```

该回调函数用于向服务器发送事件流，该回调函数需要 `requests` 库

参数

- root：该参数为根 url，回调函数将在每个 epoch 后把产生的事件流发送到该地址，事件将被发往 `root + '/publish/epoch/end/'`。发送方法为 HTTP POST，其 `data` 字段的数据是按 JSON 格式编码的事件字典。

LearningRateScheduler

```
keras.callbacks.LearningRateScheduler(schedule)
```

该回调函数是学习率调度器

参数

- schedule：函数，该函数以 epoch 号为参数（从 0 算起的整数），返回一个新学习率（浮点数）

TensorBoard

```
keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=0)
```

该回调函数是一个可视化的展示器

TensorBoard 是 TensorFlow 提供的可视化工具，该回调函数将日志信息写入 TensorBorad，使得你可以动态的观察训练和测试指标的图像以及不同层的激活值直方图。

如果已经通过 pip 安装了 TensorFlow，我们可通过下面的命令启动 TensorBoard：

```
tensorboard --logdir=/full_path_to_your_logs
```

更多的参考信息，请点击这里

参数

- log_dir：保存日志文件的地址，该文件将被 TensorBoard 解析以用于可视化
- histogram_freq：计算各个层激活值直方图的频率（每多少个 epoch 计算一次），如果设置为 0 则不计算。

ReduceLROnPlateau

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

当评价指标不在提升时，减少学习率

当学习停滞时，减少 2 倍或 10 倍的学习率常常能获得较好的效果。该回调函数检测指标的情况，如果在 `patience` 个 epoch 中看不到模型性能提升，则减少学习率

参数

- monitor：被监测的量
- factor：每次减少学习率的因子，学习率将以 `lr = lr*factor` 的形式被减少
- patience：当 patience 个 epoch 过去而模型性能不提升时，学习率减少的动作会被触发
- mode：‘auto’，‘min’，‘max’之一，在 `min` 模式下，如果检测值触发学习率减少。在 `max` 模式下，当检测值不再上升则触发学习率减少。
- epsilon：阈值，用来确定是否进入检测值的“平原区”
- cooldown：学习率减少后，会经过 cooldown 个 epoch 才重新进行正常操作
- min_lr：学习率的下限

示例：

```
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
              patience=5, min_lr=0.001)
model.fit(X_train, Y_train, callbacks=[reduce_lr])
```

CSVLogger

```
keras.callbacks.CSVLogger(filename, separator=',', append=False)
```

将 epoch 的训练结果保存在 csv 文件中，支持所有可被转换为 string 的值，包括 1D 的可迭代数值如 np.ndarray.

参数

- fiename：保存的 csv 文件名，如 `run/log.csv`
- separator：字符串，csv 分隔符
- append：默认为 False，为 True 时 csv 文件如果存在则继续写入，为 False 时总是覆盖 csv 文件

示例

```
csv_logger = CSVLogger('training.log')
model.fit(X_train, Y_train, callbacks=[csv_logger])
```

LambdaCallback

```
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)
```

用于创建简单的 callback 的 callback 类

该 callback 的匿名函数将会在适当的时候调用，注意，该回调函数假定了一些位置参数 `on_eopoch_begin` 和 `on_epoch_end` 假定输入的参数是 `epoch, logs`. `on_batch_begin` 和 `on_batch_end` 假定输入的参数是 `batch, logs`，`on_train_begin` 和 `on_train_end` 假定输入的参数是 `logs`

参数

- on_epoch_begin: 在每个 epoch 开始时调用
- on_epoch_end: 在每个 epoch 结束时调用
- on_batch_begin: 在每个 batch 开始时调用
- on_batch_end: 在每个 batch 结束时调用
- on_train_begin: 在训练开始时调用
- on_train_end: 在训练结束时调用

示例

```
# Print the batch number at the beginning of every batch.
batch_print_callback = LambdaCallback(on_batch_begin=lambda batch, logs:
print(batch))

# Plot the loss after every epoch.
import numpy as np
import matplotlib.pyplot as plt
plot_loss_callback = LambdaCallback(on_epoch_end=lambda epoch, logs:
plt.plot(np.arange(epoch), logs['loss']))
```

```
# Terminate some processes after having finished model training.
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, plot_loss_callback,
cleanup_callback])
```

编写自己的回调函数

我们可以通过继承 `keras.callbacks.Callback` 编写自己的回调函数，回调函数通过类成员 `self.model` 访问访问，该成员是模型的一个引用。

这里是一个简单的保存每个 batch 的 loss 的回调函数：

```
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'))
```

例子：记录损失函数的历史数据

```
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, init='uniform'))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

history = LossHistory()
model.fit(X_train, Y_train, batch_size=128, nb_epoch=20, verbose=0,
callbacks=[history])

print history.losses
# outputs
'''
[0.66047596406559383, 0.3547245744908703, ..., 0.25953155204159617,
0.25901699725311789]
```

例子：模型检查点

```
from keras.callbacks import ModelCheckpoint

model = Sequential()
model.add(Dense(10, input_dim=784, init='uniform'))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

'''
saves the model weights after each epoch if the validation loss decreased
```

```
'''
checkpointer = ModelCheckpoint(filepath="/tmp/weights.hdf5", verbose=1,
save_best_only=True)
model.fit(X_train, Y_train, batch_size=128, nb_epoch=20, verbose=0,
validation_data=(X_test, Y_test), callbacks=[checkpointer])
```

- 约束项

来自 `constraints` 模块的函数在优化过程中为网络的参数施加约束

惩罚项基于层进行惩罚，目前惩罚项的接口与层有关，但 `Dense, TimeDistributedDense, MaxoutDense, Covolution1D, Covolution2D, Convolution3D` 具有共同的接口。

这些层通过一下关键字施加约束项

- `W_constraint`：对主权重矩阵进行约束
- `b_constraint`：对偏置向量进行约束

```
from keras.constraints import maxnorm
model.add(Dense(64, W_constraint = maxnorm(2)))
```

预定义约束项

- maxnorm(m=2)：最大模约束
- nonneg()：非负性约束
- unitnorm()：单位范数约束, 强制矩阵沿最后一个轴拥有单位范数

- 常用数据库

CIFAR10 小图片分类数据集

该数据库具有 50,000 个 32*32 的彩色图片作为训练集，10,000 个图片作为测试集。图片一共有 10 个类别。

使用方法

```
from keras.datasets import cifar10

(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

返回值：

两个 Tuple

`X_train` 和 `X_test` 是形如（nb_samples, 3, 32, 32）的 RGB 三通道图像数据，数据类型是无符号 8 位整形（uint8）

`Y_train` 和 `Y_test` 是形如（nb_samples,）标签数据，标签的范围是 0~9

CIFAR100 小图片分类数据库

该数据库具有 50,000 个 32*32 的彩色图片作为训练集，10,000 个图片作为测试集。图片一共有 100 个类别，每个类别有 600 张图片。这 100 个类别又分为 20 个大类。

使用方法

```
from keras.datasets import cifar100

(X_train, y_train), (X_test, y_test) = cifar100.load_data(label_mode='fine')
```

参数

- label_mode：为‘fine’或‘coarse’之一，控制标签的精细度，‘fine’获得的标签是 100 个小类的标签，‘coarse’获得的标签是大类的标签

返回值

两个 Tuple,`(X_train, y_train), (X_test, y_test)`，其中

- X_train 和 X_test：是形如（nb_samples, 3, 32, 32）的 RGB 三通道图像数据，数据类型是无符号 8 位整形（uint8）
- y_train 和 y_test：是形如（nb_samples,）标签数据，标签的范围是 0~9

IMDB 影评倾向分类

本数据库含有来自 IMDB 的 25,000 条影评，被标记为正面/负面两种评价。影评已被预处理为词下标构成的序列。方便起见，单词的下标基于它在数据集中出现的频率标定，例如整数 3 所编码的词为数据集中第 3 常出现的词。这样的组织方法使得用户可以快速完成诸如“只考虑最常出现的 10,000 个词，但不考虑最常出现的 20 个词”这样的操作

按照惯例，0 不代表任何特定的词，而用来编码任何未知单词

使用方法

```
from keras.datasets import imdb

(X_train, y_train), (X_test, y_test) = imdb.load_data(path="imdb_full.pkl",
                                                      nb_words=None,
                                                      skip_top=0,
                                                      maxlen=None,
                                                      test_split=0.1)
                                                      seed=113,
                                                      start_char=1,
                                                      oov_char=2,
                                                      index_from=3)
```

参数

- path：如果你在本机上已有此数据集（位于`'~/.keras/datasets/'+path`），则载入。否则数据将下载到该目录下
- nb_words：整数或 None，要考虑的最常见的单词数，任何出现频率更低的单词将会被编码到 0 的位置。
- skip_top：整数，忽略最常出现的若干单词，这些单词将会被编码为 0
- maxlen：整数，最大序列长度，任何长度大于此值的序列将被截断
- seed：整数，用于数据重排的随机数种子
- start_char：字符，序列的起始将以该字符标记，默认为 1 因为 0 通常用作 padding
- oov_char：字符，因 `nb_words` 或 `skip_top` 限制而 cut 掉的单词将被该字符代替
- index_from：整数，真实的单词（而不是类似于 `start_char` 的特殊占位符）将从这个下标开始

返回值

两个 Tuple,`(X_train, y_train), (X_test, y_test)`，其中

- X_train 和 X_test：序列的列表，每个序列都是词下标的列表。如果指定了 `nb_words`，则序列中可能的最大下标为 `nb_words-1`。如果指定了 `maxlen`，则序列的最大可能长度为 `maxlen`
- y_train 和 y_test：为序列的标签，是一个二值 list

路透社新闻主题分类

本数据库包含来自路透社的 11,228 条新闻，分为了 46 个主题。与 IMDB 库一样，每条新闻被编码为一个词下标的序列。

使用方法

```
from keras.datasets import reuters

(X_train, y_train), (X_test, y_test) = reuters.load_data(path="reuters.pkl",
                                                         nb_words=None,
                                                         skip_top=0,
                                                         maxlen=None,
                                                         test_split=0.2,
                                                         seed=113,
                                                         start_char=1,
                                                         oov_char=2,
                                                         index_from=3)
```

参数的含义与 IMDB 同名参数相同，唯一多的参数是：`test_split`，用于指定从原数据中分割出作为测试集的比例。该数据库支持获取用于编码序列的词下标：

```
word_index = reuters.get_word_index(path="reuters_word_index.pkl")
```

上面代码的返回值是一个以单词为关键字，以其下标为值的字典。例如，
`word_index['giraffe']`的值可能为 `1234`

数据库将会被下载到`'~/.keras/datasets/'+path`

MNIST 手写数字识别

本数据库有 60,000 个用于训练的 28*28 的灰度手写数字图片，10,000 个测试图片

使用方法

```
from keras.datasets import mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

返回值

两个 Tuple,`(X_train, y_train), (X_test, y_test)`，其中

- X_train 和 X_test：是形如（nb_samples, 28, 28）的灰度图片数据，数据类型是无符号 8 位整形（uint8）
- y_train 和 y_test：是形如（nb_samples,）标签数据，标签的范围是 0~9

数据库将会被下载到`'~/.keras/datasets/'+path`

- 初始化方法

初始化方法定义了对 Keras 层设置初始化权重的方法
不同的层可能使用不同的关键字来传递初始化方法，一般来说指定初始化方法的关键字是 `init`，例如：

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

预定义初始化方法

- uniform
- lecun_uniform: 即有输入节点数之平方根放缩后的均匀分布初始化（LeCun 98）.
- normal
- identity：仅用于权值矩阵为方阵的 2D 层（`shape[0]=shape[1]`）
- orthogonal：仅用于权值矩阵为方阵的 2D 层（`shape[0]=shape[1]`），参考 Saxe et al.

- zero
- glorot_normal：由扇入扇出放缩后的高斯初始化（Glorot 2010）
- glorot_uniform
- he_normal：由扇入放缩后的高斯初始化（He et al.,2014）
- he_uniform

指定初始化方法传入的可以是一个字符串(必须与上面某种预定义方法匹配),也可以是一个可调用的对象.如果传入可调用的对象,则该对象必须包含两个参数:`shape`(待初始化的变量的 shape)和 `name`(该变量的名字),该可调用对象必须返回一个(Keras)变量,例如 `K.variable()`返回的就是这种变量,下面是例子:

```
from keras import backend as K
import numpy as np

def my_init(shape, name=None):
    value = np.random.random(shape)
    return K.variable(value, name=name)

model.add(Dense(64, init=my_init))
```

你也可以按这种方法使用 `keras.initializations` 中的函数:

```
from keras import initializations

def my_init(shape, name=None):
    return initializations.normal(shape, scale=0.01, name=name)

model.add(Dense(64, init=my_init))
```

- 性能评估

使用方法

性能评估模块提供了一系列用于模型性能评估的函数,这些函数在模型编译时由 `metrics` 关键字设置

性能评估函数类似与目标函数, 只不过该性能的评估结果讲不会用于训练.

可以通过字符串来使用域定义的性能评估函数,也可以自定义一个 Theano/TensorFlow 函数并使用之

参数

- y_true:真实标签,theano/tensorflow 张量
- y_pred:预测值, 与 y_true 形式相同的 theano/tensorflow 张量

返回值

单个用以代表输出各个数据点上均值的值

可用预定义张量

除 fbeta_score 额外拥有默认参数 beta=1 外,其他各个性能指标的参数均为 y_true 和 y_pred

- binary_accuracy: 对二分类问题,计算在所有预测值上的平均正确率
- categorical_accuracy:对多分类问题,计算再所有预测值上的平均正确率
- sparse_categorical_accuracy:与 `categorical_accuracy` 相同,在对稀疏的目标值预测时有用
- top_k_categorical_accracy: 计算 top-k 正确率,当预测值的前 k 个值中存在目标类别即认为预测正确
- mean_squared_error:计算预测值与真值的均方差
- mean_absolute_error:计算预测值与真值的平均绝对误差
- mean_absolute_percentage_error:计算预测值与真值的平均绝对误差率
- mean_squared_logarithmic_error:计算预测值与真值的平均指数误差
- hinge:计算预测值与真值的 hinge loss
- squared_hinge:计算预测值与真值的平方 hinge loss
- categorical_crossentropy:计算预测值与真值的多类交叉熵(输入值为二值矩阵,而不是向量)
- sparse_categorical_crossentropy:与多类交叉熵相同,适用于稀疏情况
- binary_crossentropy:计算预测值与真值的交叉熵
- poisson:计算预测值与真值的泊松函数值
- cosine_proximity:计算预测值与真值的余弦相似性
- matthews_correlation:计算预测值与真值的马氏距离
- precision：计算精确度，注意 percision 跟 accuracy 是不同的。percision 用于评价多标签分类中有多少个选中的项是正确的
- recall：召回率，计算多标签分类中有多少正确的项被选中
- fbeta_score:计算 F 值,即召回率与准确率的加权调和平均,该函数在多标签分类(一个样本有多个标签)时有用,如果只使用准确率作为度量,模型只要把所有输入分类为"所有类别"就可以获得完美的准确率,为了避免这种情况,度量指标应该对错误的选择进行惩罚. F-beta 分值(0 到 1 之间)通过准确率和召回率的加权调和平均来更好的度量.当 beta 为 1 时,该指标等价于 F-measure,beta<1 时,模型选对正确的标签更加重要,而 beta>1 时,模型对选错标签有更大的惩罚.
- fmeasure：计算 f-measure，即 percision 和 recall 的调和平均

定制评估函数

定制的评估函数可以在模型编译时传入,该函数应该以`(y_true, y_pred)`为参数,并返回单个张量,或从 `metric_name` 映射到 `metric_value` 的字典,下面是一个示例:

```
# for custom metrics
import keras.backend as K

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

def false_rates(y_true, y_pred):
    false_neg = ...
```

```
    false_pos = ...
    return {
        'false_neg': false_neg,
        'false_pos': false_pos,
    }

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

- 目标函数 objectives

目标函数，或称损失函数，是编译一个模型必须的两个参数之一：

```
model.compile(loss='mean_squared_error', optimizer='sgd')
```

可以通过传递预定义目标函数名字指定目标函数，也可以传递一个 Theano/TensroFlow 的符号函数作为目标函数，该函数对每个数据点应该只返回一个标量值，并以下列两个参数为参数：

- y_true：真实的数据标签，Theano/TensorFlow 张量
- y_pred：预测值，与 y_true 相同 shape 的 Theano/TensorFlow 张量

真实的优化目标函数是在各个数据点得到的损失函数值之和的均值

请参考目标实现代码获取更多信息

可用的目标函数

- mean_squared_error 或 mse
- mean_absolute_error 或 mae
- mean_absolute_percentage_error 或 mape
- mean_squared_logarithmic_error 或 msle
- squared_hinge
- hinge
- binary_crossentropy（亦称作对数损失，logloss）
- categorical_crossentropy：亦称作多类的对数损失，注意使用该目标函数时，需要将标签转化为形如`(nb_samples, nb_classes)`的二值序列
- sparse_categorical_crossentrop：如上，但接受稀疏标签。注意，使用该函数时仍然需要你的标签与输出值的维度相同，你可能需要在标签数据上增加一个维度：`np.expand_dims(y,-1)`
- kullback_leibler_divergence:从预测值概率分布 Q 到真值概率分布 P 的信息增益,用以度量两个分布的差异.
- poisson：即`(predictions - targets * log(predictions))`的均值
- cosine_proximity：即预测值与真实标签的余弦距离平均值的相反数

注意: 当使用"categorical_crossentropy"作为目标函数时,标签应该为多类模式,即 one-hot 编码的向量,而不是单个数值. 可以使用工具中的 `to_categorical` 函数完成该转换.示例如下:

```
from keras.utils.np_utils import to_categorical

categorical_labels = to_categorical(int_labels, nb_classes=None)
```

- 优化器 optimizers

优化器是编译 Keras 模型必要的两个参数之一

```
model = Sequential()
model.add(Dense(64, init='uniform', input_dim=10))
model.add(Activation('tanh'))
model.add(Activation('softmax'))

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

可以在调用 `model.compile()`之前初始化一个优化器对象，然后传入该函数（如上所示），也可以在调用 `model.compile()`时传递一个预定义优化器名。在后者情形下，优化器的参数将使用默认值。

```
# pass optimizer by name: default parameters will be used
model.compile(loss='mean_squared_error', optimizer='sgd')
```

所有优化器都可用的参数

参数 `clipnorm` 和 `clipvalue` 是所有优化器都可以使用的参数,用于对梯度进行裁剪.示例如下:

```
# all parameter gradients will be clipped to
# a maximum norm of 1.
sgd = SGD(lr=0.01, clipnorm=1.)

# all parameter gradients will be clipped to
# a maximum value of 0.5 and
# a minimum value of -0.5.
sgd = SGD(lr=0.01, clipvalue=0.5)
```

SGD

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

随机梯度下降法，支持动量参数，支持学习衰减率，支持 Nesterov 动量

参数

- lr：大于 0 的浮点数，学习率
- momentum：大于 0 的浮点数，动量参数
- decay：大于 0 的浮点数，每次更新后的学习率衰减值
- nesterov：布尔值，确定是否使用 Nesterov 动量

RMSprop

`keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-06)`

除学习率可调整外，建议保持优化器的其他默认参数不变

该优化器通常是面对递归神经网络时的一个良好选择

参数

- lr：大于 0 的浮点数，学习率
- rho：大于 0 的浮点数
- epsilon：大于 0 的小浮点数，防止除 0 错误

Adagrad

`keras.optimizers.Adagrad(lr=0.01, epsilon=1e-06)`

建议保持优化器的默认参数不变

Adagrad

- lr：大于 0 的浮点数，学习率
- epsilon：大于 0 的小浮点数，防止除 0 错误

Adadelta

`keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=1e-06)`

建议保持优化器的默认参数不变

参数

- lr：大于 0 的浮点数，学习率
- rho：大于 0 的浮点数
- epsilon：大于 0 的小浮点数，防止除 0 错误

参考文献

- Adadelta - an adaptive learning rate method

Adam

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08)
```

该优化器的默认值来源于参考文献

参数

- lr：大于 0 的浮点数，学习率
- beta_1/beta_2：浮点数， 0<beta<1，通常很接近 1
- epsilon：大于 0 的小浮点数，防止除 0 错误

参考文献

- Adam - A Method for Stochastic Optimization

Adamax

```
keras.optimizers.Adamax(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08)
```

Adamax 优化器来自于 Adam 的论文的 Section7，该方法是基于无穷范数的 Adam 方法的变体。

默认参数由论文提供

参数

- lr：大于 0 的浮点数，学习率
- beta_1/beta_2：浮点数， 0<beta<1，通常很接近 1
- epsilon：大于 0 的小浮点数，防止除 0 错误

参考文献

- Adam - A Method for Stochastic Optimization

Nadam

```
keras.optimizers.Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
schedule_decay=0.004)
```

Nesterov Adam optimizer: Adam 本质上像是带有动量项的 RMSprop，Nadam 就是带有 Nesterov 动量的 Adam RMSprop

默认参数来自于论文，推荐不要对默认参数进行更改。

参数

- lr：大于 0 的浮点数，学习率
- beta_1/beta_2：浮点数， 0<beta<1，通常很接近 1
- epsilon：大于 0 的小浮点数，防止除 0 错误

- 正则项

正则项在优化过程中层的参数或层的激活值添加惩罚项，这些惩罚项将与损失函数一起作为网络的最终优化目标
惩罚项基于层进行惩罚，目前惩罚项的接口与层有关，但 `Dense, TimeDistributedDense, MaxoutDense, Covolution1D, Covolution2D, Convolution3D` 具有共同的接口。

这些层有三个关键字参数以施加正则项：

- `W_regularizer`：施加在权重上的正则项，为 `WeightRegularizer` 对象
- `b_regularizer`：施加在偏置向量上的正则项，为 `WeightRegularizer` 对象
- `activity_regularizer`：施加在输出上的正则项，为 `ActivityRegularizer` 对象

例子

```
from keras.regularizers import l2, activity_l2
model.add(Dense(64, input_dim=64, W_regularizer=l2(0.01),
activity_regularizer=activity_l2(0.01)))
```

预定义正则项

```
keras.regularizers.WeightRegularizer(l1=0., l2=0.)

keras.regularizers.ActivityRegularizer(l1=0., l2=0.)
```

缩写

`keras.regularizers` 支持以下缩写

- l1(l=0.01)：L1 正则项，又称 LASSO
- l2(l=0.01)：L2 正则项，又称权重衰减或 Ridge
- l1l2(l1=0.01, l2=0.01)： L1-L2 混合正则项, 又称 ElasticNet
- activity_l1(l=0.01)： L1 激活值正则项
- activity_l2(l=0.01)： L2 激活值正则项
- activity_l1l2(l1=0.01, l2=0.01)： L1+L2 激活值正则项

【Tips】正则项通常用于对模型的训练施加某种约束，L1 正则项即 L1 范数约束，该约束会使被约束矩阵/向量更稀疏。L2 正则项即 L2 范数约束，该约束会使被约束的矩阵/向量更平滑，因为它对脉冲型的值有很大的惩罚。【@Bigmoyan】

- 模型可视化

`keras.utils.visualize_util` 模块提供了画出 Keras 模型的函数（利用 graphviz）
该函数将画出模型结构图，并保存成图片：

```
from keras.utils.visualize_util import plot
plot(model, to_file='model.png')
```

`plot` 接收两个可选参数：

- `show_shapes`：指定是否显示输出数据的形状，默认为 `False`
- `show_layer_names`:指定是否显示层名称,默认为 `True`

我们也可以直接获取一个 `pydot.Graph` 对象，然后按照自己的需要配置它，例如，如果要在 ipython 中展示图片

```
from IPython.display import SVG
from keras.utils.visualize_util import model_to_dot

SVG(model_to_dot(model).create(prog='dot', format='svg'))
```

【Tips】依赖 pydot-ng 和 graphviz，命令行输入 `pip install pydot-ng & brew install graphviz`

- Preprocessing
  - 图片预处理 图片生成器 ImageDataGenerator

```
keras.preprocessing.image.ImageDataGenerator(featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    rotation_range=0.,
    width_shift_range=0.,
    height_shift_range=0.,
    shear_range=0.,
    zoom_range=0.,
    channel_shift_range=0.,
    fill_mode='nearest',
    cval=0.,
    horizontal_flip=False,
    vertical_flip=False,
    rescale=None,
    dim_ordering=K.image_dim_ordering())
```

用以生成一个 batch 的图像数据，支持实时数据提升。训练时该函数会无限生成数据，直到达到规定的 epoch 次数为止。

参数

- featurewise_center：布尔值，使输入数据集去中心化（均值为 0）, 按 feature 执行
- samplewise_center：布尔值，使输入数据的每个样本均值为 0
- featurewise_std_normalization：布尔值，将输入除以数据集的标准差以完成标准化, 按 feature 执行
- samplewise_std_normalization：布尔值，将输入的每个样本除以其自身的标准差
- zca_whitening：布尔值，对输入数据施加 ZCA 白化
- rotation_range：整数，数据提升时图片随机转动的角度
- width_shift_range：浮点数，图片宽度的某个比例，数据提升时图片水平偏移的幅度
- height_shift_range：浮点数，图片高度的某个比例，数据提升时图片竖直偏移的幅度
- shear_range：浮点数，剪切强度（逆时针方向的剪切变换角度）
- zoom_range：浮点数或形如`[lower,upper]`的列表，随机缩放的幅度，若为浮点数，则相当于`[lower,upper] = [1 - zoom_range, 1+zoom_range]`
- channel_shift_range：浮点数，随机通道偏移的幅度
- fill_mode：； ‘constant’， ‘nearest’， ‘reflect’或 ‘wrap’之一，当进行变换时超出边界的点将根据本参数给定的方法进行处理
- cval：浮点数或整数，当 `fill_mode=constant` 时，指定要向超出边界的点填充的值
- horizontal_flip：布尔值，进行随机水平翻转
- vertical_flip：布尔值，进行随机竖直翻转
- rescale: 重放缩因子,默认为 None. 如果为 None 或 0 则不进行放缩,否则会将该数值乘到数据上(在应用其他变换之前)
- dim_ordering： ‘tf’和 ‘th’之一，规定数据的维度顺序。 ‘tf’模式下数据的形状为 `samples, height, width, channels`， ‘th’下形状为`(samples, channels, height, width)`.该参数的默认值是 Keras 配置文件`~/.keras/keras.json` 的 `image_dim_ordering` 值,如果你从未设置过的话,就是'tf'

方法

- fit(X, augment=False, rounds=1)：计算依赖于数据的变换所需要的统计信息(均值方差等),只有使用 `featurewise_center，featurewise_std_normalization` 或 `zca_whitening` 时需要此函数。
    - X：numpy array，样本数据，秩应为 4.在黑白图像的情况下 channel 轴的值为 1，在彩色图像情况下值为 3

    - augment：布尔值，确定是否使用随即提升过的数据
    - round：若设`augment=True`，确定要在数据上进行多少轮数据提升，默认值为 1
    - seed: 整数,随机数种子
- flow(self, X, y, batch_size=32, shuffle=True, seed=None, save_to_dir=None, save_prefix='', save_format='jpeg')：接收 numpy 数组和标签为参数,生成经过数据提升或标准化后的 batch 数据,并在一个无限循环中不断的返回 batch 数据
    - X：样本数据，秩应为 4.在黑白图像的情况下 channel 轴的值为 1，在彩色图像情况下值为 3
    - y：标签
    - batch_size：整数，默认 32
    - shuffle：布尔值，是否随机打乱数据，默认为 True
    - save_to_dir：None 或字符串，该参数能让你将提升后的图片保存起来，用以可视化
    - save_prefix：字符串，保存提升后图片时使用的前缀, 仅当设置了`save_to_dir`时生效
    - save_format："png"或"jpeg"之一，指定保存图片的数据格式,默认"jpeg"
    - yields:形如(x,y)的 tuple,x 是代表图像数据的 numpy 数组.y 是代表标签的 numpy 数组. 该迭代器无限循环.
    - seed: 整数,随机数种子
- flow_from_directory(directory): 以文件夹路径为参数,生成经过数据提升/归一化后的数据,在一个无限循环中无限产生 batch 数据
    - directory: 目标文件夹路径,对于每一个类,该文件夹都要包含一个子文件夹.子文件夹中任何 JPG、PNG 和 BNP 的图片都会被生成器使用.详情请查看此脚本
    - target_size: 整数 tuple,默认为(256, 256). 图像将被 resize 成该尺寸
    - color_mode: 颜色模式,为"grayscale","rgb"之一,默认为"rgb".代表这些图片是否会被转换为单通道或三通道的图片.
    - classes: 可选参数,为子文件夹的列表,如['dogs','cats']默认为 None. 若未提供,则该类别列表将自动推断(类别的顺序将按照字母表顺序映射到标签值)
    - class_mode: "categorical", "binary", "sparse"或 None 之一. 默认为"categorical. 该参数决定了返回的标签数组的形式, "categorical"会返回 2D 的 one-hot 编码标签,"binary"返回 1D 的二值标签."sparse"返回 1D 的整数标签,如果为 None 则不返回任何标签, 生成器将仅仅生成 batch 数据, 这种情况在使用`model.predict_generator()`和`model.evaluate_generator()`等函数时会用到.
    - batch_size: batch 数据的大小,默认 32
    - shuffle: 是否打乱数据,默认为 True

- seed: 可选参数,打乱数据和进行变换时的随机数种子
- save_to_dir: None 或字符串，该参数能让你将提升后的图片保存起来，用以可视化
- save_prefix：字符串，保存提升后图片时使用的前缀, 仅当设置了 `save_to_dir` 时生效
- save_format："png"或"jpeg"之一，指定保存图片的数据格式,默认"jpeg"
- flollow_links: 是否访问子文件夹中的软链接

例子

使用`.flow()`的例子

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

datagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True)

# compute quantities required for featurewise normalization
# (std, mean, and principal components if ZCA whitening is applied)
datagen.fit(X_train)

# fits the model on batches with real-time data augmentation:
model.fit_generator(datagen.flow(X_train, Y_train, batch_size=32),
                    samples_per_epoch=len(X_train), nb_epoch=nb_epoch)

# here's a more "manual" example
for e in range(nb_epoch):
    print 'Epoch', e
    batches = 0
    for X_batch, Y_batch in datagen.flow(X_train, Y_train, batch_size=32):
        loss = model.train(X_batch, Y_batch)
        batches += 1
        if batches >= len(X_train) / 32:
            # we need to break the loop by hand because
            # the generator loops indefinitely
            break
```

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,
        samples_per_epoch=2000,
        nb_epoch=50,
        validation_data=validation_generator,
        nb_val_samples=800)
```

同时变换图像和 mask

```
# we create two instances with the same arguments
data_gen_args = dict(featurewise_center=True,
                     featurewise_std_normalization=True,
                     rotation_range=90.,
                     width_shift_range=0.1,
                     height_shift_range=0.1,
                     zoom_range=0.2)
image_datagen = ImageDataGenerator(**data_gen_args)
mask_datagen = ImageDataGenerator(**data_gen_args)

# Provide the same seed and keyword arguments to the fit and flow methods
seed = 1
image_datagen.fit(images, augment=True, seed=seed)
mask_datagen.fit(masks, augment=True, seed=seed)

image_generator = image_datagen.flow_from_directory(
    'data/images',
    class_mode=None,
    seed=seed)

mask_generator = mask_datagen.flow_from_directory(
    'data/masks',
    class_mode=None,
    seed=seed)

# combine generators into one which yields image and masks
train_generator = zip(image_generator, mask_generator)

model.fit_generator(
    train_generator,
    samples_per_epoch=2000,
    nb_epoch=50)
```

- 序列预处理

• 填充序列 pad_sequences

```
keras.preprocessing.sequence.pad_sequences(sequences, maxlen=None,
dtype='int32', padding='pre', truncating='pre', value=0)
```

将长为 `nb_samples` 的序列（标量序列）转化为形如`(nb_samples,nb_timesteps)`2D numpy array。如果提供了参数 `maxlen`，`nb_timesteps=maxlen`，否则其值为最长序列的长度。其他短于该长度的序列都会在后部填充 0 以达到该长度。长于 `nb_timesteps` 的序列将会被截断，以使其匹配目标长度。padding 和截断发生的位置分别取决于 `padding` 和 `truncating`.

参数

- sequences：浮点数或整数构成的两层嵌套列表
- maxlen：None 或整数，为序列的最大长度。大于此长度的序列将被截短，小于此长度的序列将在后部填 0.
- dtype：返回的 numpy array 的数据类型
- padding： ‘pre’或‘post’，确定当需要补 0 时，在序列的起始还是结尾补
- truncating： ‘pre’或‘post’，确定当需要截断序列时，从起始还是结尾截断
- value：浮点数，此值将在填充时代替默认的填充值 0

返回值

返回形如`(nb_samples,nb_timesteps)`的 2D 张量

跳字 skipgrams

```
keras.preprocessing.sequence.skipgrams(sequence, vocabulary_size,
    window_size=4, negative_samples=1., shuffle=True,
    categorical=False, sampling_table=None)
```

skipgrams 将一个词向量下标的序列转化为下面的一对 tuple：

- 对于正样本，转化为（word，word in the same window）
- 对于负样本，转化为（word，random word from the vocabulary）

【Tips】根据维基百科，n-gram 代表在给定序列中产生连续的 n 项，当序列句子时，每项就是单词，此时 n-gram 也称为 shingles。而 skip-gram 的推广，skip-gram 产生的 n 项子序列中，各个项在原序列中不连续，而是跳了 k 个字。例如，对于句子：

“the rain in Spain falls mainly on the plain”

其 2-grams 为子序列集合：

the rain，rain in，in Spain，Spain falls，falls mainly，mainly on，on the，the plain

其 1-skip-2-grams 为子序列集合：

the in, rain Spain, in falls, Spain mainly, falls on, mainly the, on plain.

更多详情请参考 Efficient Estimation of Word Representations in Vector Space【@BigMoyan】

参数

- sequence：下标的列表，如果使用 sampling_tabel，则某个词的下标应该为它在数据库中的顺序。（从 1 开始）

- vocabulary_size：整数，字典大小
- window_size：整数，正样本对之间的最大距离
- negative_samples：大于 0 的浮点数，等于 0 代表没有负样本，等于 1 代表负样本与正样本数目相同，以此类推（即负样本的数目是正样本的 `negative_samples` 倍）
- shuffle：布尔值，确定是否随机打乱样本
- categorical：布尔值，确定是否要使得返回的标签具有确定类别
- sampling_table：形如`(vocabulary_size,)`的 numpy array，其中`sampling_table[i]`代表没有负样本或随机负样本。等于 1 为与正样本的数目相同 采样到该下标为 i 的单词的概率（假定该单词是数据库中第 i 常见的单词）

输出

函数的输出是一个`(couples,labels)`的元组，其中：

- `couples` 是一个长为 2 的整数列表：`[word_index,other_word_index]`
- `labels` 是一个仅由 0 和 1 构成的列表，1 代表 `other_word_index` 在 `word_index` 的窗口，0 代表 `other_word_index` 是词典里的随机单词。
- 如果设置 `categorical` 为 `True`，则标签将以 one-hot 的方式给出，即 1 变为[0,1]，0 变为[1,0]

获取采样表 make_sampling_table

```
keras.preprocessing.sequence.make_sampling_table(size, sampling_factor=1e-5)
```

该函数用以产生 `skipgrams` 中所需要的参数 `sampling_table`。这是一个长为 `size` 的向量，`sampling_table[i]`代表采样到数据集中第 i 常见的词的概率（为平衡期起见，对于越经常出现的词，要以越低的概率采到它）

参数

- size：词典的大小
- sampling_factor：此值越低，则代表采样时更缓慢的概率衰减（即常用的词会被以更低的概率被采到），如果设置为 1，则代表不进行下采样，即所有样本被采样到的概率都是 1。

- 文本预处理

句子分割 text_to_word_sequence

```
keras.preprocessing.text.text_to_word_sequence(text,
    filters=base_filter(), lower=True, split=" ")
```

本函数将一个句子拆分成单词构成的列表

参数

- text：字符串，待处理的文本
- filters：需要滤除的字符的列表或连接形成的字符串，例如标点符号。默认值为`base_filter()`，包含标点符号，制表符和换行符等
- lower：布尔值，是否将序列设为小写形式
- split：字符串，单词的分隔符，如空格

返回值

字符串列表

---

one-hot 编码

```
keras.preprocessing.text.one_hot(text, n,
    filters=base_filter(), lower=True, split=" ")
```

本函数将一段文本编码为 one-hot 形式的码，即仅记录词在词典中的下标。

【Tips】 从定义上，当字典长为 n 时，每个单词应形成一个长为 n 的向量，其中仅有单词本身在字典中下标的位置为 1，其余均为 0，这称为 one-hot。【@Bigmoyan】

为了方便起见，函数在这里仅把“1”的位置，即字典中词的下标记录下来。

参数

- n：整数，字典长度

返回值

整数列表，每个整数是[1,n]之间的值，代表一个单词（不保证唯一性，即如果词典长度不够，不同的单词可能会被编为同一个码）。

分词器 Tokenizer

```
keras.preprocessing.text.Tokenizer(nb_words=None, filters=base_filter(),
    lower=True, split=" ")
```

Tokenizer 是一个用于向量化文本，或将文本转换为序列（即单词在字典中的下标构成的列表，从 1 算起）的类。

构造参数

- 与`text_to_word_sequence`同名参数含义相同

- nb_words：None 或整数，处理的最大单词数量。若被设置为整数，则分词器将被限制为处理数据集中最常见的 `nb_words` 个单词

类方法

- fit_on_texts(texts)
    - texts：要用以训练的文本列表
- texts_to_sequences(texts)
    - texts：待转为序列的文本列表
    - 返回值：序列的列表，列表中每个序列对应于一段输入文本
- texts_to_sequences_generator(texts)
    - 本函数是 `texts_to_sequences` 的生成器函数版
    - texts：待转为序列的文本列表
    - 返回值：每次调用返回对应于一段输入文本的序列
- texts_to_matrix(texts, mode)：
    - texts：待向量化的文本列表
    - mode：‘binary’，‘count’，‘tfidf’，‘freq’之一，默认为‘binary’
    - 返回值：形如`(len(texts), nb_words)`的 numpy array
- fit_on_sequences(sequences):
    - sequences：要用以训练的序列列表
- sequences_to_matrix(sequences):
    - sequences：待向量化的序列列表
    - mode：‘binary’，‘count’，‘tfidf’，‘freq’之一，默认为‘binary’
    - 返回值：形如`(len(sequences), nb_words)`的 numpy array

属性

- word_counts:字典，将单词（字符串）映射为它们在训练期间出现的次数。仅在调用 fit_on_texts 之后设置。
- word_docs: 字典，将单词（字符串）映射为它们在训练期间所出现的文档或文本的数量。仅在调用 fit_on_texts 之后设置。
- word_index: 字典，将单词（字符串）映射为它们的排名或者索引。仅在调用 fit_on_texts 之后设置。
- document_count: 整数。分词器被训练的文档（文本或者序列）数量。仅在调用 fit_on_texts 或 fit_on_sequences 之后设置。

- Utils
    - 数据工具

get_file

```
get_file(fname, origin, untar=False, md5_hash=None, cache_subdir='datasets')
```

从给定的 URL 中下载文件, 可以传递 MD5 值用于数据校验(下载后或已经缓存的数据均可)

参数

- fname: 文件名
- origin: 文件的 URL 地址
- untar: 布尔值,是否要进行解压
- md5_hash: MD5 哈希值,用于数据校验
- cache_subdir: 用于缓存数据的文件夹

返回值

下载后的文件地址

- I/O 工具

HDF5 矩阵

```
keras.utils.io_utils.HDF5Matrix(datapath, dataset, start=0, end=None,
normalizer=None)
```

这是一个使用 HDF5 数据集代替 Numpy 数组的方法。

例子

```
X_data = HDF5Matrix('input/file.hdf5', 'data')
model.predict(X_data)
```

提供 start 和 end 参数可以使用数据集的切片。

可选的，可以给出归一化函数（或 lambda 表达式）。 这将在每个检索的数据集的切片上调用。

参数

- datapath：字符串，HDF5 文件的路径
- dataset：字符串，在 datapath 中指定的文件中的 HDF5 数据集的名称
- start：整数，指定数据集的所需切片的开始
- end：整数，指定数据集的所需切片的结尾
- normalizer：数据集在被检索时的调用函数

- Keras 层工具

layer_from_config

```
layer_from_config(config, custom_objects={})
```

从配置生成 Keras 层对象

参数

- config:形如{'class_name':str, 'config':dict}的字典
- custom_objects: 字典,用以将定制的非 Keras 对象之类名/函数名映射为类/函数对象

返回值

层对象,包含 Model,Sequential 和其他 Layer

- numpy 工具

to_categorical

```
to_categorical(y, nb_classes=None)
```

将类别向量(从 0 到 nb_classes 的整数向量)映射为二值类别矩阵, 用于应用到以 `categorical_crossentropy` 为目标函数的模型中.

参数

- y: 类别向量
- nb_classes:总共类别数

convert_kernel

```
convert_kernel(kernel, dim_ordering='default')
```

将卷积核矩阵(numpy 数组)从 Theano 形式转换为 Tensorflow 形式,或转换回来(该转化时自可逆的)

- Models
  - 关于 Keras 模型 repeated
  - 函数式模型接口

为什么叫“函数式模型”，请查看“Keras 新手指南”的相关部分

Keras 的函数式模型为 `Model`，即广义的拥有输入和输出的模型，我们使用 `Model` 来初始化一个函数式模型

```
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)
```

在这里，我们的模型以 a 为输入，以 b 为输出，同样我们可以构造拥有多输入和多输出的模型

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

常用 Model 属性

- `model.layers`：组成模型图的各个层
- `model.inputs`：模型的输入张量列表
- `model.outputs`：模型的输出张量列表

Model 模型方法

compile

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

本函数编译模型以供训练，参数有

- optimizer：优化器，为预定义优化器名或优化器对象，参考优化器
- loss：损失函数，为预定义损失函数名或一个目标函数，参考损失函数
- metrics：列表，包含评估模型在训练和测试时的性能的指标，典型用法是`metrics=['accuracy']`如果要在多输出模型中为不同的输出指定不同的指标，可像该参数传递一个字典，例如`metrics={'ouput_a': 'accuracy'}`
- sample_weight_mode：如果你需要按时间步为样本赋权（2D 权矩阵），将该值设为“temporal”。默认为“None”，代表按样本赋权（1D 权）。如果模型有多个输出，可以向该参数传入指定 sample_weight_mode 的字典或列表。在下面`fit`函数的解释中有相关的参考内容。
- weighted_metrics: metrics 列表，在训练和测试过程中，这些 metrics 将由`sample_weight`或`clss_weight`计算并赋权
- target_tensors: 默认情况下，Keras 将为模型的目标创建一个占位符，该占位符在训练过程中将被目标数据代替。如果你想使用自己的目标张量（相应的，Keras 将不会在训练时期望为这些目标张量载入外部的 numpy 数据），你可以通过该参数手动指定。目标张量可以是一个单独的张量（对应于单输出模型），也可以是一个张量列表，或者一个 name->tensor 的张量字典。
- kwargs：使用 TensorFlow 作为后端请忽略该参数，若使用 Theano/CNTK 作为后端，kwargs 的值将会传递给 K.function。如果使用 TensorFlow 为后端，这里的值会被传给 tf.Session.run

当为参数传入非法值时会抛出异常

【Tips】如果你只是载入模型并利用其 predict，可以不用进行 compile。在 Keras 中，compile 主要完成损失函数和优化器的一些配置，是为训练服务的。predict 会在内部进行符号函数的编译工作（通过调用_make_predict_function 生成函数），

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)
```

本函数用以训练模型，参数有：

- x：输入数据。如果模型只有一个输入，那么 x 的类型是 numpy array，如果模型有多个输入，那么 x 的类型应当为 list，list 的元素是对应于各个输入的 numpy array。如果模型的每个输入都有名字，则可以传入一个字典，将输入名与其输入数据对应起来。
- y：标签，numpy array。如果模型有多个输出，可以传入一个 numpy array 的 list。如果模型的输出拥有名字，则可以传入一个字典，将输出名与其标签对应起来。
- batch_size：整数，指定进行梯度下降时每个 batch 包含的样本数。训练时一个 batch 的样本会被计算一次梯度下降，使目标函数优化一步。
- epochs：整数，训练终止时的 epoch 值，训练将在达到该 epoch 值时停止，当没有设置 initial_epoch 时，它就是训练的总轮数，否则训练的总轮数为 epochs - inital_epoch
- verbose：日志显示，0 为不在标准输出流输出日志信息，1 为输出进度条记录，2 为每个 epoch 输出一行记录
- callbacks：list，其中的元素是 `keras.callbacks.Callback` 的对象。这个 list 中的回调函数将会在训练过程中的适当时机被调用，参考回调函数
- validation_split：0~1 之间的浮点数，用来指定训练集的一定比例数据作为验证集。验证集将不参与训练，并在每个 epoch 结束后测试的模型的指标，如损失函数、精确度等。注意，validation_split 的划分在 shuffle 之后，因此如果你的数据本身是有序的，需要先手工打乱再指定 validation_split，否则可能会出现验证集样本不均匀。
- validation_data：形式为（X，y）或（X，y，sample_weights）的 tuple，是指定的验证集。此参数将覆盖 validation_spilt。
- shuffle：布尔值，表示是否在训练过程中每个 epoch 前随机打乱输入样本的顺序。
- class_weight：字典，将不同的类别映射为不同的权值，该参数用来在训练过程中调整损失函数（只能用于训练）。该参数在处理非平衡的训练数据（某些类的训练样本数很少）时，可以使得损失函数对样本数不足的数据更加关注。
- sample_weight：权值的 numpy array，用于在训练时调整损失函数（仅用于训练）。可以传递一个 1D 的与样本等长的向量用于对样本进行 1 对 1 的加权，或者在面对时序数据时，传递一个的形式为（samples，sequence_length）的矩阵来为每个时间步上的样本赋不同的权。这种情况下请确定在编译模型时添加了 `sample_weight_mode='temporal'`。
- initial_epoch: 从该参数指定的 epoch 开始训练，在继续之前的训练时有用。

- steps_per_epoch: 一个 epoch 包含的步数（每一步是一个 batch 的数据送入），当使用如 TensorFlow 数据 Tensor 之类的输入张量进行训练时，默认的 None 代表自动分割，即数据集样本数/batch 样本数。
- validation_steps: 仅当 steps_per_epoch 被指定时有用，在验证集上的 step 总数。

输入数据与规定数据不匹配时会抛出错误

`fit` 函数返回一个 `History` 的对象，其 `History.history` 属性记录了损失函数和其他指标的数值随 epoch 变化的情况，如果有验证集的话，也包含了验证集的这些指标变化情况

---

evaluate

```
evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=None)
```

本函数按 batch 计算在某些输入数据上模型的误差，其参数有：

- x：输入数据，与 `fit` 一样，是 numpy array 或 numpy array 的 list
- y：标签，numpy array
- batch_size：整数，含义同 `fit` 的同名参数
- verbose：含义同 `fit` 的同名参数，但只能取 0 或 1
- sample_weight：numpy array，含义同 `fit` 的同名参数

本函数返回一个测试误差的标量值（如果模型没有其他评价指标），或一个标量的 list（如果模型还有其他的评价指标）。`model.metrics_names` 将给出 list 中各个值的含义。

如果没有特殊说明，以下函数的参数均保持与 `fit` 的同名参数相同的含义

如果没有特殊说明，以下函数的 verbose 参数（如果有）均只能取 0 或 1

---

predict

```
predict(self, x, batch_size=32, verbose=0)
```

本函数按 batch 获得输入数据对应的输出，其参数有：

函数的返回值是预测值的 numpy array

---

train_on_batch

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

本函数在一个 batch 的数据上进行一次参数更新

函数返回训练误差的标量值或标量值的 list，与 evaluate 的情形相同。

---

test_on_batch

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

本函数在一个 batch 的样本上对模型进行评估

函数的返回与 evaluate 的情形相同

---

predict_on_batch

```
predict_on_batch(self, x)
```

本函数在一个 batch 的样本上对模型进行测试

函数返回模型在一个 batch 上的预测结果

---

fit_generator

```
fit_generator(self, generator, steps_per_epoch, epochs=1, verbose=1,
callbacks=None, validation_data=None, validation_steps=None, class_weight=None,
max_q_size=10, workers=1, pickle_safe=False, initial_epoch=0)
```

利用 Python 的生成器，逐个生成数据的 batch 并进行训练。生成器与模型将并行执行以提高效率。例如，该函数允许我们在 CPU 上进行实时的数据提升，同时在 GPU 上进行模型训练

函数的参数是：

- generator：生成器函数，生成器的输出应该为：
    - 一个形如（inputs，targets）的 tuple
    - 一个形如（inputs, targets,sample_weight）的 tuple。所有的返回值都应该包含相同数目的样本。生成器将无限在数据集上循环。每个 epoch 以经过模型的样本数达到 `samples_per_epoch` 时，记一个 epoch 结束
- steps_per_epoch：整数，当生成器返回 `steps_per_epoch` 次数据时计一个 epoch 结束，执行下一个 epoch
- epochs：整数，数据迭代的轮数
- verbose：日志显示，0 为不在标准输出流输出日志信息，1 为输出进度条记录，2 为每个 epoch 输出一行记录
- validation_data：具有以下三种形式之一
    - 生成验证集的生成器
    - 一个形如（inputs,targets）的 tuple
    - 一个形如（inputs,targets，sample_weights）的 tuple

- validation_steps: 当 validation_data 为生成器时，本参数指定验证集的生成器返回次数
- class_weight：规定类别权重的字典，将类别映射为权重，常用于处理样本不均衡问题。
- sample_weight：权值的 numpy array，用于在训练时调整损失函数（仅用于训练）。可以传递一个 1D 的与样本等长的向量用于对样本进行 1 对 1 的加权，或者在面对时序数据时，传递一个的形式为（samples，sequence_length）的矩阵来为每个时间步上的样本赋不同的权。这种情况下请确定在编译模型时添加了 `sample_weight_mode='temporal'`。
- workers：最大进程数
- max_q_size：生成器队列的最大容量
- pickle_safe: 若为真，则使用基于进程的线程。由于该实现依赖多进程，不能传递 non picklable（无法被 pickle 序列化）的参数到生成器中，因为无法轻易将它们传入子进程中。
- initial_epoch: 从该参数指定的 epoch 开始训练，在继续之前的训练时有用。

函数返回一个 `History` 对象

例子

```
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
        x1, x2, y = process_line(line)
        yield ({'input_1': x1, 'input_2': x2}, {'output': y})
    f.close()

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

---

evaluate_generator

```
evaluate_generator(self, generator, steps, max_q_size=10, workers=1,
pickle_safe=False)
```

本函数使用一个生成器作为数据源，来评估模型，生成器应返回与 `test_on_batch` 的输入数据相同类型的数据。

函数的参数是：

- generator：生成输入 batch 数据的生成器
- val_samples：生成器应该返回的总样本数
- steps：生成器要返回数据的轮数
- max_q_size：生成器队列的最大容量
- nb_worker：使用基于进程的多线程处理时的进程数

- pickle_safe：若设置为 True，则使用基于进程的线程。注意因为它的实现依赖于多进程处理，不可传递不可 pickle 的参数到生成器中，因为它们不能轻易的传递到子进程中。

---

predict_generator

```
predict_generator(self, generator, steps, max_queue_size=10, workers=1,
use_multiprocessing=False, verbose=0)
```

从一个生成器上获取数据并进行预测，生成器应返回与 `predict_on_batch` 输入类似的数据

函数的参数是：

- generator：生成输入 batch 数据的生成器
- val_samples：生成器应该返回的总样本数
- max_q_size：生成器队列的最大容量
- nb_worker：使用基于进程的多线程处理时的进程数
- pickle_safe：若设置为 True，则使用基于进程的线程。注意因为它的实现依赖于多进程处理，不可传递不可 pickle 的参数到生成器中，因为它们不能轻易的传递到子进程中。
    - Sequential 模型接口 repeated

al below repeated, pass.

- Other
    - 激活函数 Activations
    - Application 应用
    - 回调函数 Callbacks
    - 约束项
    - 常用数据库
    - 初始化方法
    - 性能评估
    - 目标函数 objectives
    - 优化器 optimizers
    - 正则项
    - 模型可视化
- Preprocessing
    - 图片预处理
    - 序列预处理
    - 文本预处理
        - 文本预处理
            - 句子分割 text_to_word_sequence
            - one-hot 编码

- 特征哈希 hashing_trick
- 分词器 Tokenizer