Keras(无 CNN 层版)

整理人：Anthony_Yu

2020/06/13

目录

1. 引言

简易和快速的原型设计（keras 具有高度模块化，极简，和可扩充特性）,支持 CNN 和 RNN 或二者的结合，支持任意的链接方案（包括多输入和多输出训练,无缝 CPU 和 GPU 切换。

1.1keras 的设计原理

（1） 模块性: 模型可理解为一个独立的序列或图，完全可配置的模块以最少的代价自由组合在一起。具体而言，网络层、损失函数、优化器、初始化策略、激活函数、正则化方法都是独立的模块，我们可以使用它们来构建自己的模型
（2）极简主义: 每个模块都应该尽量的简洁。每一段代码都应该在初次阅读时都显得直观易懂。没有黑魔法，因为它将给迭代和创新带来麻烦
（3）易扩展性: 添加新模块超级简单的容易，只需要仿照现有的模块编写新的类或函数即可。创建新模块的便利性使得 Keras 更适合于先进的研究工作
（4）与 Python 协作: Keras 没有单独的模型配置文件类型，模型由 python 代码描述，使其更紧凑和更易 debug，并提供了扩展的便利性

1..2keras 的简单介绍

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

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

然后，将一些网络层通过.add()堆叠起来，就构成了一个模型：

```
from keras.layers import Dense, Activation
model.add(Dense(output_dim=64, input_dim=100))
model.add(Activation("relu"))
model.add(Dense(output_dim=10))
model.add(Activation("softmax"))
```

完成模型的搭建后，我们需要使用.compile()方法来编译模型：

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

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

```
from keras.optimizers import SGD
```

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

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

model.fit(X_train, Y_train, nb_epoch=5, batch_size=32)

当然，我们也可以手动将一个个 batch 的数据送入网络中训练，这时候需要使用

model.train_on_batch(X_batch, Y_batch)

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

loss_and_metrics = model.evaluate(X_test, Y_test, batch_size=32)

又或者，可以使用的模型，对新的数据进行预测

classes = model.predict_classes(X_test, batch_size=32)
proba = model.predict_proba(X_test, batch_size=32)

2. 一些基本概念

2.1 关于符号计算

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

2.2 关于张量层

使用这个词汇的目的是为了表述统一，张量可以看作是向量、矩阵的自然推广，我们用张量来表示广泛的数据类型规模最小的张量是 0 阶张量，即标量，也就是一个数，而当我们把一些数有序的排列起来，就形成了 1 阶张量，也就是一个向量，但是如果我们继续把一组向量有序的排列起来，就形成了 2 阶张量，也就是说一个矩阵把矩阵摞起来，就是 3 阶张量，我们可以称为一个立方体，具有 3 个颜色通道的彩色图片就是一个这样的立方体张量的阶数有时候也称为维度，或者轴，轴这个词翻译自英文 axis。譬如一个矩阵[[1,2],[3,4]]，是一个 2 阶张量，有两个维度或轴，沿着第 0 个轴（为了与 python 的计数方式一致，本文档维度和轴从 0 算起）你看到的是[1,2]，[3,4]两个向量，沿着第 1 个轴你看到的是[1,3]，[2,4]两个向量。例子如下：

```
import numpy as np
a = np.array([[1,2],[3,4]])
sum0 = np.sum(a, axis=0)
sum1 = np.sum(a, axis=1)
print sum0
print sum1
```

2.3 泛型模型

在原本的 Keras 版本中，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；

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

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

相关链接：http://keras-cn.readthedocs.io/en/latest/getting_started/concepts/

3. Sequential 模型

Sequential 是多个网络层的线性堆叠

第一种，可以通过向 Sequential 模型传递一个 layer 的 list 来构造该模型，例子如下：

```
from keras.models import Sequential
from keras.layers import Dense, Activation
model = Sequential([
Dense(32, input_dim=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：(推荐初学者)

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

3.1 指定输入数据的 shape

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

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

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

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

3.1.1 三种严格等价的方法

下面的三个指定输入数据 shape 的方法是严格等价的：

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

model = Sequential()
model.add(Dense(32, batch_input_shape=(None, 784)))
# note that batch dimension is "None" here,
# so the model will be able to process batches of any size.</pre>

model = Sequential()
model.add(Dense(32, input_dim=784))#输出维度 32，输入维度 784
```

3.1.2LSTM 中三种严格等价的方法

下面三种方法也是严格等价的：

```
model = Sequential()
model.add(LSTM(32, input_shape=(10, 64)))

model = Sequential()
model.add(LSTM(32, batch_input_shape=(None, 10, 64)))

model = Sequential()
```

model.add(LSTM(32, input_length=10, input_dim=64))

3.2Merge 层

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

```
from keras.layers import Merge

left_branch = Sequential()
left_branch.add(Dense(32, input_dim=784))

right_branch = Sequential()
right_branch.add(Dense(32, input_dim=784))

merged = Merge([left_branch, right_branch], mode='concat')

final_model = Sequential()
final_model.add(merged)
final_model.add(Dense(10, activation='softmax'))
```

当然在 Merge 层中也支持一些预定义的合并模式，包括如下：

sum(defualt):逐元素相加

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

mul：逐元素相乘

ave：张量平均

dot：张量相乘，可以通过 dot_axis 关键字参数来指定要消去的轴

cos：计算 2D 张量（即矩阵）中各个向量的余弦距离

在模型编译时也可以通过这两个分支的模型进行编译和训练，代码如下：

```
final_model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
final_model.fit([input_data_1, input_data_2], targets)  # 为每个模型输入传递一个数据数组
```

当然也可以为 Merge 层提供关键字参数 mode，以实现任意的变换，例如：

```
merged = Merge([left_branch, right_branch], mode=lambda x: x[0] - x[1])
```

对于不能通过 Sequential 和 Merge 组合生成的复杂模型，可以参考 2.3 泛型模型，也可以去 Keras 中文文档进行相关 API 的了解。

3.3 模型的编译

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

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

2. 损失函数 loss：该参数为模型试图最小化的目标函数，它可为预定义的损失函数名，如 categorical_crossentropy、mse，也可以为一个损失函数
3. 指标列表 metrics：对分类问题，我们一般将该列表设置为 metrics=['accuracy']。指标可以是一个预定义指标的名字,也可以是一个用户定制的函数.指标函数应该返回单个张量,或一个完成 metric_name - > metric_value 映射的字典。

3.3.1 集中应对不同实验的不同方法

```
# for a multi-class classification problem(对于多级分类问题)
model.compile(optimizer='rmsprop',loss='categorical_crossentropy',metrics=['accuracy'])

# for a binary classification problem(对于二进制分类问题)
model.compile(optimizer='rmsprop',loss='binary_crossentropy',metrics=['accuracy'])

# for a mean squared error regression problem(均方误差回归问题)
model.compile(optimizer='rmsprop',loss='mse')

# for custom metrics(用于自定义指标)
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])
```

3.4 模型训练

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，例子如下：

3.4.1 例子一（二分类）：

```
# for a single-input model with 2 classes (binary)(对一个简单的二分类输入模块)
model = Sequential()
model.add(Dense(1, input_dim=784, activation='sigmoid'))
model.compile(optimizer='rmsprop',loss='binary_crossentropy',metrics=['accuracy'])

# generate dummy data(生成虚拟数据)
import numpy as np
data = np.random.random((1000, 784))
labels = np.random.randint(2, size=(1000, 1))

# train the model, iterating on the data in batches(训练模型，在 32 个样本中分批迭代数据)
# of 32 samples
model.fit(data, labels, nb_epoch=10, batch_size=32)
```

3.4.2 例子二（多分类）

```
# for a multi-input model with 10 classes:（用于具有 10 个类别的多输入模型）

left_branch = Sequential()
left_branch.add(Dense(32, input_dim=784))

right_branch = Sequential()
right_branch.add(Dense(32, input_dim=784))

merged = Merge([left_branch, right_branch], mode='concat')

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

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

# generate dummy data(生成虚拟数据)
import numpy as np
from keras.utils.np_utils import to_categorical
data_1 = np.random.random((1000, 784))
data_2 = np.random.random((1000, 784))
```

```
# these are integers between 0 and 9(取 0 到 9 之间的整数)
labels = np.random.randint(10, size=(1000, 1))
# we convert the labels to a binary matrix of size (1000, 10)
# for use with categorical_crossentropy(用于分类交叉熵)
labels = to_categorical(labels, 10)

# train the model
# note that we are passing a list of Numpy arrays as training data(注意一下正在传递 Numpy 数组的列表作为训练数据)
# since the model has 2 inputs(两个模型的输入)
model.fit([data_1, data_2], labels, nb_epoch=10, batch_size=32)
```

3.5 一些例子

3.5.1 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

model = Sequential()
# Dense(64) 是具有 64 个隐藏单元的完全连接层。
#在第一层中，必须指定预期的输入数据形状：
#这里是 20 维向量。
model.add(Dense(64, input_dim=20, init='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(64, init='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(10, init='uniform'))
model.add(Activation('softmax'))

sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(X_train, y_train,
          nb_epoch=20,
```

```
            batch_size=16)
score = model.evaluate(X_test, y_test, batch_size=16)
```

3.5.2 相似 MLP 的另一种实现

```
model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

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

3.5.3 用于二分类的多层感知器

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

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

3.5.4 类似 VGG 的卷积神经网络

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.optimizers import SGD

model = Sequential()
# 输入：输入 100x100 的具有 3 各通道的图像-> (3, 100, 100) tensors.
```

```
# 在这里，应用32 个每个大小为3x3 的卷积滤波器。
model.add(Convolution2D(32, 3, 3, border_mode='valid', input_shape=(3, 100, 100)))
model.add(Activation('relu'))
model.add(Convolution2D(32, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Convolution2D(64, 3, 3, border_mode='valid'))
model.add(Activation('relu'))
model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
#注意：在这里 Keras 会自动进行形状推断。
model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(10))
model.add(Activation('softmax'))

sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

model.fit(X_train, Y_train, batch_size=32, nb_epoch=1)
```

3.5.5 使用 LSTM 的序列分类

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

model = Sequential()
model.add(Embedding(max_features, 256, input_length=maxlen))
model.add(LSTM(output_dim=128, activation='sigmoid', inner_activation='hard_sigmoid'))
model.add(Dropout(0.5))
```

```
model.add(Dense(1))
model.add(Activation('sigmoid'))

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

model.fit(X_train, Y_train, batch_size=16, nb_epoch=10)
score = model.evaluate(X_test, Y_test, batch_size=16)
```

3.6.6 用于序列分类的栈式 LSTM

说明：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
nb_classes = 10

# 接受的数据类型: (batch_size, timesteps, data_dim)
model = Sequential()
model.add(LSTM(32, return_sequences=True,input_shape=(timesteps, data_dim)))  # 返回
size=32 的向量序列
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'])

#构建模拟训练集
x_train = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, nb_classes))

# 构建模拟验证集
x_val = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, nb_classes))
```

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

3.5.7 采用状态 LSTM 的相同模型

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

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

data_dim = 16
timesteps = 8
nb_classes = 10
batch_size = 32

#接受的数据类型: (batch_size, timesteps, data_dim)
# 请注意，由于网络是有状态的，所以必须提供完整的 batch_input_shape.
# 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'])

# 构建模拟训练集
x_train = np.random.random((batch_size * 10, timesteps, data_dim))
y_train = np.random.random((batch_size * 10, nb_classes))

# 构建模拟验证集
x_val = np.random.random((batch_size * 3, timesteps, data_dim))
y_val = np.random.random((batch_size * 3, nb_classes))

model.fit(x_train, y_train,
```

```
        batch_size=batch_size, nb_epoch=5,
        validation_data=(x_val, y_val))
```

3.5.8 将两个 LSTM 合并作为编码端来处理两路序列的分类

说明：两路输入序列通过两个 LSTM 被编码为特征向量两路特征向量被串连在一起，然后通过一个全连接网络得到结果。

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

data_dim = 16
timesteps = 8
nb_classes = 10

encoder_a = Sequential()
encoder_a.add(LSTM(32, input_shape=(timesteps, data_dim)))

encoder_b = Sequential()
encoder_b.add(LSTM(32, input_shape=(timesteps, data_dim)))

decoder = Sequential()
decoder.add(Merge([encoder_a, encoder_b], mode='concat'))
decoder.add(Dense(32, activation='relu'))
decoder.add(Dense(nb_classes, activation='softmax'))

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

# 构建模拟训练集
x_train_a = np.random.random((1000, timesteps, data_dim))
x_train_b = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, nb_classes))

# 构建模拟验证集
x_val_a = np.random.random((100, timesteps, data_dim))
x_val_b = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, nb_classes))

decoder.fit([x_train_a, x_train_b], y_train,
            batch_size=64, nb_epoch=5,
```

```
validation_data=([x_val_a, x_val_b], y_val))
```

相关链接：
http://www.jianshu.com/p/9dc9f41f0b29
http://keras-cn.readthedocs.io/en/latest/getting_started/sequential_model/
https://github.com/kylibin28/kerasLSTM/blob/master/classifiers/LSTM.py
https://blog.csdn.net/loseinvain/article/details/79642721

4. 泛型模型

Keras 泛型模型接口是用户定义多输出模型、非循环有向模型或具有共享层的模型等复杂模型的途径。
1. 对于层对象接受张量为参数，将返回一个张量。张量在数学上只是数据结构的扩充，一阶张量就是向量，二阶张量就是矩阵，三阶张量就是立方体。在这里张量只是广义的表达一种数据结构，例如一张彩色图像其实就是一个三阶张量(每一阶都是 one-hot 向量)，它由三个通道的像素值堆叠而成。而 10000 张彩色图构成的一个数据集合则是四阶张量。
2. 输入是张量，输出也是张量的一个框架就是一个模型
3. 这样的模型可以被像 Keras 的 Sequential 一样被训练。以下面的全连接层为例：

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

# 返回一个张量
inputs = Input(shape=(784,))

# 层实例可在张量上调用，并返回张量
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# 创建一个模型，其中模型中包括一个输入层和三个全连接层
model = Model(input=inputs, output=predictions)
model.compile(optimizer='rmsprop',loss='categorical_crossentropy',metrics=['accuracy'])
model.fit(data, labels)  # 开始训练
```

4.1 相关解释

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,))
#起作用，并返回我们上面定义的 10 给 softmax
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

# 输入张量用于 20 个时间步长的序列，
# 每个包含 784 维向量
input_sequences = Input(shape=(20, 784))

# 这会将先前的模型应用于输入序列中的每个时间步。
# 先前模型的输出是 10 给 softmax，
# 因此下一层的输出将是 20 个大小为 10 的向量的序列。
processed_sequences = TimeDistributed(model)(input_sequences)
```

4.2 多输入和多输出模型

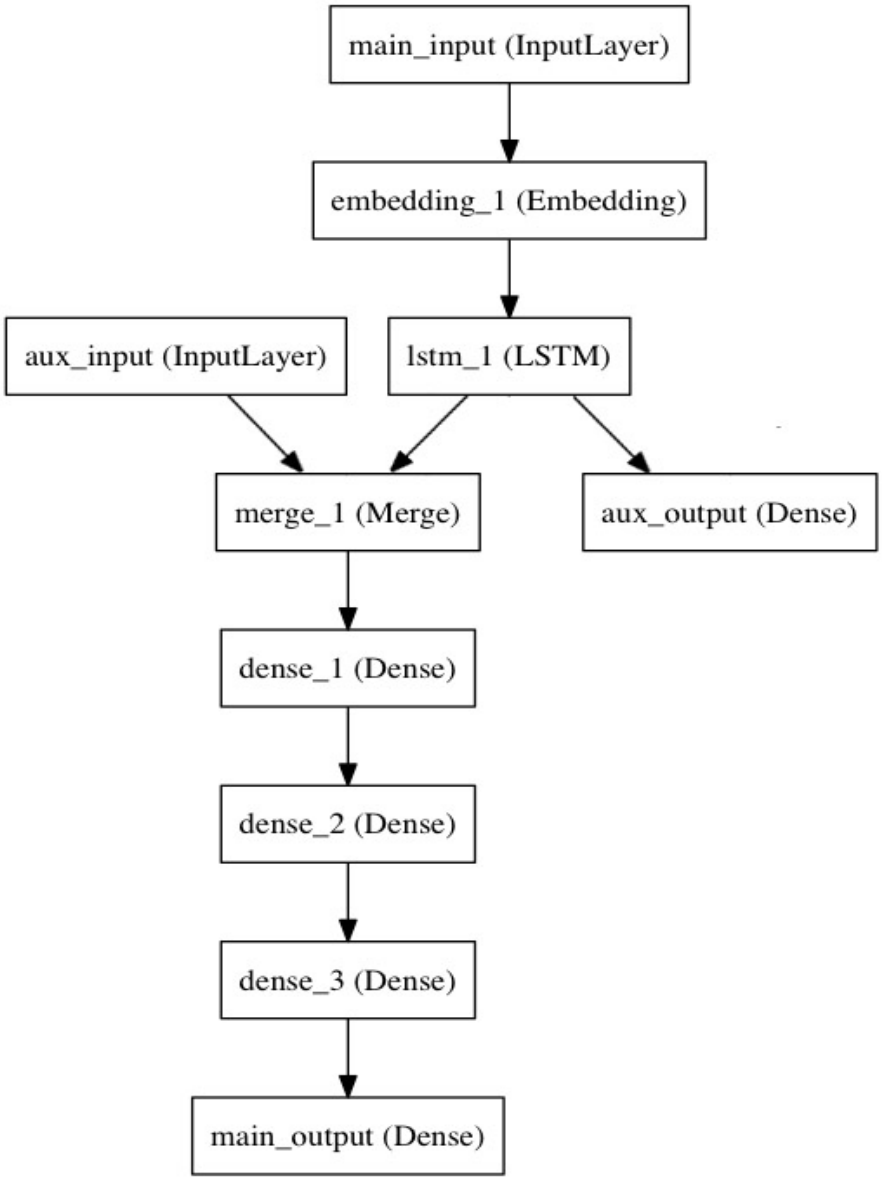
说明：使用泛型模型的一个典型场景是搭建多输入、多输出的模型。考虑这样一个模型。我们希望预测 Twitter 上一条新闻会被转发和点赞多少次。模型的主要输入是新闻本身，也就是一个词语的序列。但我们还可以拥有额外的输入，如新闻发布的日期等。这个模型的损失函数将由两部分组成，辅助的损失函数评估仅仅基于新闻本身做出预测的情况，主损失函数评估基于新闻和额外信息的预测的情况，即使来自主损失函数的梯度发生弥散，来自辅助损失函数的信息也能够训练 Embeddding 和 LSTM 层。在模型中早点使用主要的损失函数是对于深度网络的一个良好的正则方法。总而言之，该模型框图如下：

```
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)
```



可以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

# 输入：用于接收 100 个整数（介于 1 和 10000 之间）的序列。
# 注意：我们可以通过为任何图层传递“名称”参数来命名。
main_input = Input(shape=(100,), dtype='int32', name='main_input')

# 该嵌入层将对输入序列进行编码
# 变成一系列密集的 512 维向量。
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)
```

```
# LSTM 会将向量序列转换为单个向量，
# 包含有关整个序列的信息
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)
```

紧接着，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')

# 在顶部堆叠一个深度的全连接网络
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
#最后添加主要的逻辑回归层
main_output = Dense(1, activation='sigmoid', name='main_output')(x)
```

最后，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)
```

4.3 共享层

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

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

tweet_a = Input(shape=(140, 256))
tweet_b = Input(shape=(140, 256))
```

若要对不同的输入共享同一层，就初始化该层一次，然后多次调用它

```
# 该层可以作为矩阵输入
# 并将返回大小为 64 的向量
shared_lstm = LSTM(64)

# 当我们重用同一个图层实例时
# 多次，图层的权重
# 也被重用
#（实际上是* the same *层）
encoded_a = shared_lstm(tweet_a)
encoded_b = shared_lstm(tweet_b)

# 连接两个向量
merged_vector = merge([encoded_a, encoded_b], mode='concat', concat_axis=-1)

# 然后在顶部加一个逻辑回归
predictions = Dense(1, activation='sigmoid')(merged_vector)

#我们定义了一个可训练的模型，将 tweet 输入预测
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)
```

4.4 层“节点”的概念及相关例子

说明：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...

4.4.1 inception 模型

```
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
```

4.4.2 卷积层的残差连接(Residual Network)

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

# 输入一个 3 通道 256*256 的图像
x = Input(shape=(3, 256, 256))
# 具有 3 个输出通道的 3x3 转换（与输入通道相同）
y = Convolution2D(3, 3, 3, border_mode='same')(x)
```

```
# 返回 x+y.
z = merge([x, y], mode='sum')
```

4.4.3. 共享视觉模型

说明：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

# 定义视觉模块
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)

# 定义数字通话模块
digit_a = Input(shape=(1, 27, 27))
digit_b = Input(shape=(1, 27, 27))

# 视觉模型将被共享，权重和所有
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)
```

4.4.4 视觉问答模型(问题性图像验证码)

说明：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，所以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

# 使用顺序模型定义视觉模型

# 该模型会将图像编码为矢量。

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())

#通过我们的视觉模型的输出得到张量：

image_input = Input(shape=(3, 224, 224))
encoded_image = vision_model(image_input)

#定义语言模型以将问题编码为向量，每个问题最长为100 个字，bingqie 将字词索引为 1

到 9999 之间的整数。

question_input = Input(shape=(100,), dtype='int32')
embedded_question = Embedding(input_dim=10000, output_dim=256,
input_length=100)(question_input)
encoded_question = LSTM(256)(embedded_question)

# 连接问题向量和图像向量：：

merged = merge([encoded_question, encoded_image], mode='concat')

# 在顶部训练超过 1000 个单词的逻辑回归

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

```
vqa_model = Model(input=[image_input, question_input], output=output)
```

4.4.5 视频问答模型

说明：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))
#这是通过先前训练的 vision_model 进行视频编码的（权重已重用），输出将是矢量序列
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)

#输入一个问题
video_question_input = Input(shape=(100,), dtype='int32')
encoded_video_question = question_encoder(video_question_input)

# 回答模块
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)
```

相关链接：http://wiki.jikexueyuan.com/project/tensorflow-zh/resources/dims_types.html

5. 常用层

5.1Dense 层

说明：Dense 就是常用的全连接层

```
keras.layers.core.Dense(
    output_dim,
    init='glorot_uniform',
    activation='linear',
    weights=None,
    W_regularizer=None,
    b_regularizer=None,
    activity_regularizer=None,
    W_constraint=None,
    b_constraint=None,
    bias=True,
    input_dim=None
)
```

对上面参数的解释：

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

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

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

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

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

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

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

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

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

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

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

注：在第一层之后，您不再需要指定输入的大小。

5.2 Activation 层

说明：激活层对一个层的输出施加激活函数

---

keras.layers.core.Activation(activation)

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

5.3 Dropout 层

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

---

keras.layers.core.Dropout(p)

p：0~1 的浮点数，控制需要断开的链接的比例

5.4 flatten 层

说明：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

---

keras.layers.core.Flatten()

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

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

5.5 Reshape 层

说明：Reshape 层用来将输入 shape 转换为特定的 shape

keras.layers.core.Reshape(target_shape)

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

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

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

5.6 Permute 层

说明：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

keras.layers.core.Permute(dims)

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

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

5.7 RepeatVector 层

说明：RepeatVector 层将输入重复 n 次

keras.layers.core.RepeatVector(n)

n：整数，重复的次数

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

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

5.8 Merge 层

说明：Merge 层根据给定的模式，将一个张量列表中的若干张量合并为一个单独的张量

```
keras.engine.topology.Merge(
    layers=None,
    mode='sum',
    concat_axis=-1,
    dot_axes=-1,
    output_shape=None,
    node_indices=None,
    tensor_indices=None,
    name=None
)
```

关于上面参数的解释：

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

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

如果为字符串，则必须是下列值之一：

"sum", "mul", "concat", "ave", "cos", "dot"

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

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

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

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

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

5.9 Lambda 层

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

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

上面参数的解释：

---

1. function：要实现的函数，该函数仅接受一个变量，即上一层的输出

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

3. `arguments`：可选，字典，用来记录向函数中传递的其他关键字参数

5.10 ActivityRegularizer 层

说明：经过本层的数据不会有任何变化，但会基于其激活值更新损失函数值

---

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

l1：1 范数正则因子（正浮点数）

l2：2 范数正则因子（正浮点数）

5.11 Masking 层

说明：使用给定的值对输入的序列信号进行“屏蔽”，用以定位需要跳过的时间步
对于输入张量的时间步，即输入张量的第 1 维度（维度从 0 开始算），如果输入张量在该时间步上都等于 mask_value，则该时间步将在模型接下来的所有层（只要支持 masking）被跳过（屏蔽）。但是，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。

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) )
```

5.12 Highway 层

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

```
keras.layers.core.Highway(
    init='glorot_uniform',
    transform_bias=-2,
    activation='linear',
    weights=None,
    W_regularizer=None,
```

b_regularizer=None,
activity_regularizer=None,
W_constraint=None,
b_constraint=None,
bias=True,
input_dim=None
)

上面参数的解释：

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

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

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

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

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

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

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

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

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

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

11.input_dim：整数，输入数据的维度。当该层作为网络的第一层时，必须指定该参数或 12.input_shape 参数。

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

5.13 MaxoutDense 层

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

相关链接：https://keras-cn.readthedocs.io/en/latest/layers/core_layer/

https://keras.io/zh/

6 递归层 Recurrent

6.1 Recurrent 层

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

---

keras.layers.recurrent.Recurrent(
weights=None,
return_sequences=False,
go_backwards=False,
stateful=False,
unroll=False,
consume_less='cpu',
input_dim=None,
input_length=None
)

对于上面参数的解释：

---

1. weights：numpy array 的 list，用以初始化权重。该 list 形如[(input_dim,

output_dim),(output_dim, output_dim),(output_dim,)]

2. return_sequences：布尔值，默认 False，控制返回类型。若为 True 则返回整个序列，否

则仅返回输出序列的最后一个输出

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

4. stateful：布尔值，默认为 False，若为 True，则一个 batch 中下标为 i 的样本的最终状态

将会用作下一个 batch 同样下标的样本的初始状态。

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

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

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

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

6.2 SimpleRNN 层

说明：全连接 RNN 网络，RNN 的输出会被回馈到输入。

```
keras.layers.recurrent.SimpleRNN(
    output_dim,
    init='glorot_uniform',
    inner_init='orthogonal',
    activation='tanh',
    W_regularizer=None,
    U_regularizer=None,
    b_regularizer=None,
    dropout_W=0.0,
    dropout_U=0.0
)
```

对上面参数的解释：

1.output_dim：内部投影和输出的维度

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

3.inner_init：内部单元的初始化方法

4.activation：激活函数，为预定义的激活函数名（参考激活函数）

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

6.U_regularizer：施加在递归权重上的正则项，为 WeightRegularizer 对象

7.b_regularizer：施加在偏置向量上的正则项，为 WeightRegularizer 对象

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

9.dropout_U：0~1 之间的浮点数，控制输入单元到递归连接的断开比例

6.3 GRU 层

说明：门限递归单元。

```
keras.layers.recurrent.GRU(
    output_dim,
    init='glorot_uniform',
    inner_init='orthogonal',
    activation='tanh',
    inner_activation='hard_sigmoid',
    W_regularizer=None,
    U_regularizer=None,
    b_regularizer=None,
    dropout_W=0.0,
    dropout_U=0.0
)
```

对上面参数的解释：

1.output_dim：内部投影和输出的维度

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

3.inner_init：内部单元的初始化方法

4.activation：激活函数，为预定义的激活函数名（参考激活函数）

5.inner_activation：内部单元激活函数

6.W_regularizer：施加在权重上的正则项，为 WeightRegularizer 对象

7.U_regularizer：施加在递归权重上的正则项，为 WeightRegularizer 对象

8.b_regularizer：施加在偏置向量上的正则项，为 WeightRegularizer 对象

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

10.dropout_U：0~1 之间的浮点数，控制输入单元到递归连接的断开比例

6.4 LSTM 层

说明：Keras 中的长短期记忆模型

```
keras.layers.recurrent.LSTM(
    output_dim,
    init='glorot_uniform',
    inner_init='orthogonal',
    forget_bias_init='one',
    activation='tanh',
    inner_activation='hard_sigmoid',
    W_regularizer=None,
    U_regularizer=None,
    b_regularizer=None,
    dropout_W=0.0,
    dropout_U=0.0
)
```

对上面参数的解释：

1.output_dim：内部投影和输出的维度

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

3.inner_init：内部单元的初始化方法

4.forget_bias_init：遗忘门偏置的初始化函数，Jozefowicz et al.建议初始化为全 1 元素

5.activation：激活函数，为预定义的激活函数名（参考激活函数）

6.inner_activation：内部单元激活函数

7.W_regularizer：施加在权重上的正则项，为 WeightRegularizer 对象

8.U_regularizer：施加在递归权重上的正则项，为 WeightRegularizer 对象

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

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

11.dropout_U：0~1 之间的浮点数，控制输入单元到递归连接的断开比例

6.4.1 训练 lstm 的参数

```
def train_lstm(
    dim_proj=72,  # word embeding dimension and LSTM number of hidden units.
    patience=15,  # Number of epoch to wait before early stop if no progress
    max_epochs=5000,  # The maximum number of epoch to run
    dispFreq=10,  # Display to stdout the training progress every N updates
    decay_c=0.,  # Weight decay for the classifier applied to the U weights.
    lrate=0.01,  # Learning rate for sgd (not used for adadelta and rmsprop)
    optimizer=adadelta,  # sgd, adadelta and rmsprop available, sgd very hard to use, not recommanded (probably need momentum and decaying learning rate).
    #  optimizer = sgd,
    encoder='lstm',  # TODO: can be removed must be lstm.
    saveto='lstm_model_best.npz',  # The best model will be saved there
    validFreq=10,  # Compute the validation error after this number of update.
    saveFreq=10,  # Save the parameters after every saveFreq updates
    maxlen=2,  # Sequence longer then this get ignored
    batch_size=64,  # The batch size during training.
    valid_batch_size=50,  # The batch size used for validation/test set.
```

```
    dataset=datanamee,
    modal_costs = [0.1, 0.1] ,# the cost for each modal
    # model_lens = model_len,
    # Parameter for extra option
    noise_std=0.,
    use_dropout=True,    # if False slightly faster, but worst test error
                         # This frequently need a bigger model.
    reload_model=None,    # Path to a saved model we want to start from.
    test_size=-1,    # If >0, we keep only this number of test example.
    max_costs = 50, # max cost for each instance to use
):
```

相关链接：https://keras-cn.readthedocs.io/en/latest/layers/recurrent_layer/

7.Embedding 层

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

```
keras.layers.embeddings.Embedding(
    input_dim,
    output_dim,
    init='uniform',
    input_length=None,
    W_regularizer=None,
    activity_regularizer=None,
    W_constraint=None,
    mask_zero=False,
    weights=None,
    dropout=0.0
)
```

对上面参数的解释：

1.input_dim：大或等于 0 的整数，字典长度，即输入数据最大下标+1

2.output_dim：大于 0 的整数，代表全连接嵌入的维度

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

数。该参数仅在不传递 weights 参数时有意义。

4.weights：权值，为 numpy array 的 list。该 list 应仅含有一个如（input_dim,output_dim）的权重矩阵

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

6.W_constraints：施加在权重上的约束项，为 Constraints 对象

7.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

8.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。

9.dropout：0~1 的浮点数，代表要断开的嵌入比例

相关链接：https://keras-cn.readthedocs.io/en/latest/layers/embedding_layer/