我的Keras使用总结（1）——Keras概述与常见问题整理

完整代码及其数据，请移步小编的GitHub

传送门：请点击我

如果点击有误：https://github.com/LeBron-Jian/DeepLearningNote

今天整理了自己所写的关于Keras的博客，有没发布的，有发布的，但是整体来说是有点乱的。上周有空，认真看了一周Keras的中文文档，稍有心得，整理于此。这里附上Keras官网地址：

Keras英文文档：https://keras.io/#installationKeras

Keras中文文档：https://keras.io/zh/

下面回顾一下自己以前写的有关Keras的博客：

Python机器学习笔记：利用Keras进行分类预测

Python机器学习笔记：使用Keras进行回归预测

首先是上面两篇博客，这两篇写出来，其实是整理笔记，单纯的学习一下基础的回归和分类，使用Keras创建一个简单的回归问题的神经网络模型和多类分类的模型。使用了最经典的回归数据波士顿房价数据集和分类数据鸢尾花数据集。创建一个简单的模型进行训练。其重点是学习基础的分类回归思想，顺便使用别人的Keras代码进行实现。

深入学习Keras中Sequential模型及方法

其次是这篇博客，当时学习Keras的时候对序贯模型不懂，就做了一个简单的笔记对此进行整理。主要整理了序贯模型的函数及其参数的意义，包括使用Keras进行模型训练的部分简单的过程，最后整理了Keras官网文档的几个例子。此时也只是对Keras有个大概的了解。

Keras深度学习之卷积神经网络（CNN）

然后上面这篇博客还是学习卷积神经网络，为什么上面这篇博客起这个一个怪异的名字呢，我当时知道自己不太懂，所以我希望自己懂了后回顾一下。而且这篇文章我没有发布，我将其从随笔改为文章，就是说是自己摘抄的笔记，希望懂后，回首一下，也算是做了个交代。

这篇杂文主要是以学习卷积神经网络为主，回顾了典型的CNN，常用的CNN框架，和Keras如何实现CNN等。主要实现了两件事情，一个是使用Keras搭建了一个卷积神经网络，一个是学习了卷积神经网络实现的过程。内容以摘抄为主，但是摘抄的确实是好文，讲解细腻，循序渐进，值得去看。

Python机器学习笔记：深入理解Keras中序贯模型和函数模型

然后就是这篇博客了，我是先学习的sklearn，然后再学习的tensorflow，Keras。所以对照着sklearn的机器学习流程学习Keras的使用流程。然后对序贯模型和函数模型进行详细学习和对比，最后整理了Keras如何保存模型。相对来说我的思路还是比较清晰了。会使用Keras搭建模型了。

那么说完了这么多，为什么还要写呢?

其实从18年九月开始，自己的机器学习之路就开始了，当时前途一片迷茫，学到哪里算哪里。到19年一月又写了几篇博客，当然后面也写了只是没有发布而已，到现在已经是20年三月了。啰嗦这么多，就是感觉自己走了弯路，让看到博客的人能少走点弯路，而且最主要的是自己整理回顾。

首先说起Keras，都知道它是基于Theano和TensorFlow的深度学习库，所以这里我们先说一说TensorFlow，上面聊过，我是先学习sklearn，再学习的TensorFlow的。通过对比sklearn的训练过程，学习Keras的训练过程。其实大同小异，那么同在哪里，异又在哪里?

上面博客只是简单的学习了一下其两者的机器学习使用流程和区别，这里从根上回顾一下机器学习和深度学习的区别。这里内容我就不写自己的拙见了，直接拿网友现成的东西（https://www.zhihu.com/question/53740695/answer/284428668）

1，机器学习和深度学习的总结

首先我们看sklearn和TensorFlow的区别。 这个问题其实等价于：现在深度学习那么火，那么是否还有必要学习传统的机器学习方法。

理论上来说，深度学习技术也是机器学习的一个组成部分，学习其他传统的机器学习方法对深入理解深度学习技术有很大的帮助，知道模型凸的条件，才能更好的理解神经网络的非凸，知道传统模型的优点，才能更好的理解深度学习并不是万能的，也有很多问题和场景直接使用深度学习方法会遇到瓶颈和问题，需要传统方法来解决。

从实践上来说，深度学习方法一般需要大量的GPU机器，工业界哪怕大公司的GPU资源也是有限的，一般只有深度学习方法效果远好于传统方法并且对业务提升很大的情况下，才会考虑使用深度学习方法，例如语音识别，图像识别等任务现在深度学习方法用的比较多，而NPL领域除了机器翻译以外，其他大部分任务仍然更常使用传统方法，传统方法一般有着更好的可解释性，这对检查调试模型也是非常有帮助的。工业上一般喜欢招能解决问题的人，而不是掌握最火技术的人，因此在了解深度学习技术的同时，学习一下传统的方法是很有好处的。

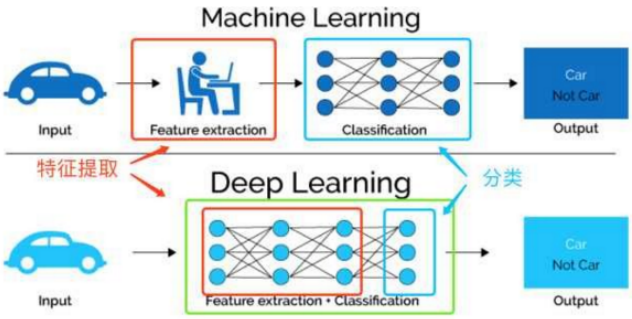
1.1 Sklearn和TensorFlow的区别

Scikit-learn（Sklearn）的定位是通过机器学习库，而TensorFlow（tf）的定位主要是深度学习库。一个显而易见的不同：tf并未提供sklearn那种强大的特征工程，如维度压缩，特征选择等。

传统机器学习：利用特征工程（feature enginerring），人为对数据进行提炼清洗

深度学习：利用表示学习（representation learning），机器学习模型自身对数据进行提炼

sklearn更倾向于使用者可以自行对数据进行处理，比如选择特征，压缩维度，转换格式，是传统机器学库，而以tf为代表的深度学习库会自动从数据中抽取有效特征，而不需要人为的来做这件事情，因此并未提供类似的功能。



上面这幅图直观的对比了我们提到的两种对于数据的学习方法，传统的机器学习方法主要依赖人工特征处理与提取，而深度学习依赖模型自身去学习数据的表示。

1.2 模型封装的抽象化程度不同，给予使用者自由度不同

sklearn中的模块都是高度抽象化，所有的分类器基本都可以在3~5行内完成，所有的转换器（如scaler和transformer）也有固定的格式，这种抽象化限制了使用者的自由度，但是增加了模型的效率，降低了批量化，标准化的难度。

比如svm分类器：

```
clf = svm.SVC() # 初始化一个分类器
clf.fit(X_train, y_train) # 训练分类器
y_predict = clf.predict(X_test) # 使用训练好的分类器进行预测<div class="open_grepper_editor" title="Edit & Save To G
```

而tf不同，虽然是深度学习库，但是它由很高的自由度。你依然可以使用它做传统机器学习所做的事情，代码是你需要自己实现算法。因此用tf类比sklearn不适合，封装在tf等工具库的Keras更像深度学习界的sklearn。

从自由度来看，tf更高；而从抽象化，封装程度来看，sklearn更高；从易用性角度来看，sklearn更高。

1.3 深度的群体，项目不同

sklearn主要适合中小型的，实用机器学习项目，尤其是那种数据量不大且需要使用者手动对数据进行处理，并选择合适模型的项目，这类项目往往在CPU上就可以完成，对硬件要求低。

tf主要适合已经明确了解需要用深度学习，且数据处理需求不高的项目。这类项目往往数据量较大，且最终需要的精度更高，一般都需要GPU加速运算。对于深度学习做"小样"可以在采样的小数据集上用Keras做快速的实验。

下面我们看一个使用Keras搭建的网络，代码如下：

```
model = Sequential() # 定义模型
model.add(Dense(units=64, activation='relu', input_dim=100)) # 定义网络结构
model.add(Dense(units=10, activation='softmax')) # 定义网络结构
model.compile(loss='categorical_crossentropy', # 定义loss函数、优化方法、评估标准
              optimizer='sgd',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, batch_size=32) # 训练模型
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128) # 评估模型
classes = model.predict(x_test, batch_size=128) # 使用训练好的数据进行预测
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

不难看出，sklearn和tf还是有很大的区别，虽然sklearn中也有神经网络模块，但是做大型的深度学习是不可能依靠sklearn的。

1.4 sklearn和TensorFlow结合使用

更常见的情况，可以把sklearn和tf，甚至Keras结合起来使用。sklearn肩负基本的数据清理任务，Keras用于对问题进行小规模试验验证想法，而tf用于在完整的数据上进行严肃的调参任务。

而单独把sklearn拿出来看的话，它的文档做的特别好，初学者跟着看一遍sklearn支持的功能就可以大概对机器学习包括的很多内容有了基本的了解。举个简单的例子，sklearn很多时候对单独的知识点有概述，比如简单的异常检测。因此，sklearn不仅仅是简单的工具库，它的文档更像是一份简单的新手入门指南。

因此，以sklearn为代表的传统机器学习库和以tf为代表的自由灵活更具有针对性的深度学习库都是机器学习者必须要了解的工具。

1. python 生成器和迭代器有这
2. 战争热诚的python全栈开发之
3. Python机器学习笔记：异常点
e Class SVM(41)
4. Python机器学习笔记：随机梯
5. 实现text-detection-ctpn一路

推荐排行榜

1. python 生成器和迭代器有这
2. 深入学习卷积神经网络中卷积
义(32)
3. Python机器学习笔记：sklear
4. 如何为开发项目编写规范的R
dows），此文详解(27)
5. 战争热诚的python全栈开发之

最新评论

1. Re:Python机器学习笔记：朴
请问datingTestSet.txt是什么数
的数据吗?

2. Re:卷积神经网络学习笔记—
networks（孪生神经网络）
请问，孪生网络是如何解决多分

3. Re:OpenCV计算机视觉学习
间几何变换（图像缩放，图像旋
图像平移，仿射变换，镜像变换
*是"近"

4. Re:python 浅析模块，包及其
学习了，多谢

5. Re:Python机器学习笔记：随
你好 想问问那个得分的代码跑出
回事呀

那么如何结合sklearn库和Keras模型做机器学习任务呢?

Keras是Python中比较流行的深度学习库，但是Keras本身关注的是深度学习。而Python中的scikit-learn库是建立在Scipy上的，有着比较有效的数值计算能力。Sklearn是一个具有全特征的通用性机器学习库，它提供了很多在深度学习中可以用到的工具，举个例子：

- 1，可以用sklearn中的 K-fold 交叉验证方法来对模型进行评估
- 2，模型参数的评估和寻找

Keras提供了深度学习模型的简便包装，可以在Sklearn中被用来做分类和回归，在本文中我们举这么一个例子：使用Keras建立神经网络分类器——KerasClassifier，并在scikit-learn库中使用这个分类器对UCI的Pima Indians数据集进行分类。

利用Keras进行分类或者回归，主要利用Keras中两个类，一个是KerasClassifier，另一个是KerasRegressor。这两个类有参数build_fn。build_fn是你创建的Keras名称，在创建一个Keras模型时，务必要把完成模型的定义，编译和返回。在这里我们假设建立的模型叫做create_model()，则：

```
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
    model.add(Dense(8, init='uniform', activation='relu'))
    model.add(Dense(1, init='uniform', activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

将建立好的模型通过参数build_fn传递到KerasClassifier中，并且定义其他的参数选项nb_epoch = 150,batch_size = 10 。KerasClassifier会自动调用fit方法。

在Sklearn中，我们使用它cross_validation的包中的StratifiedKFold来进行10折交叉验证，使用cross_val_score来对模型进行评价。

```
kfold = StratifiedKFold(y=Y, n_folds=10, shuffle=True, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

总结来说，机器学习和深度学习均需要学习，只会调用工具包的程序员不是好的机器学习者。

扯了这么多，我们明白sklearn和keras结合使用，相得益彰。

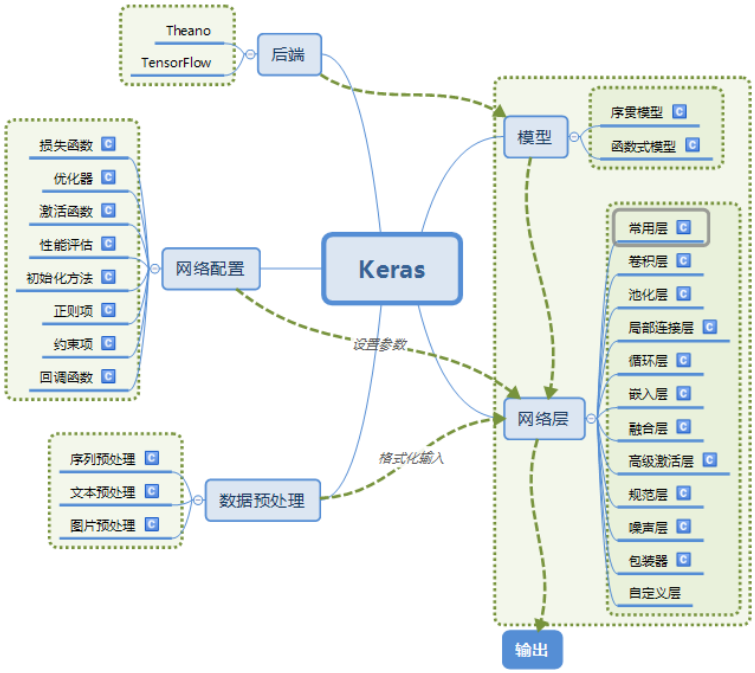
2，Keras的概述

而我之前的笔记主要的针对点就是序贯模型和卷积神经网络这块，那今天我们不学习这些东西。从架构上整体的了解一下Keras。

Keras是Python中一个以CNTK、TensorFlow或者Theano为计算后台的深度学习建模环境。相对于常见的几种深度学习计算软件，比如TensorFlow、Theano、Caffe、CNTK、Torch等，Keras在实际应用中有如下几个显著的优点。Keras在设计时以人为本，强调快速建模，用户能快速地将所需模型的结构映射到Keras代码中，尽可能减少编写代码的工作量，特别是对于成熟的模型类型，从而加快开发速度。支持现有的常见结构，比如卷积神经网络、时间递归神经网络等，足以应对大量的常见应用场景。高度模块化，用户几乎能够任意组合各个模块来构造所需的模型。

在Keras中，任何神经网络模型都可以被描述为一个图模型或者序列模型，其中的部件被划分为以下模块：神经网络层、损失函数、激活函数、初始化方法、正则化方法、优化引擎。这些模块可以以任意合理地方式放入图模型或者序列模型中来构造所需的模型，用户并不需要知道每个模块后面的细节。这种方式相比其他软件需要用户编写大量代码或者用特定语言来描述神经网络结构的方法效率高很多，也不容易出错。基于Python，用户也可以使用Python代码来描述模型，因此易用性、可扩展性都非常高。用户可以非常容易地编写自己的定制模块，或者对已有模块进行修改或者扩展，因此可以非常方便地开发和应用新的模型与方法，加快迭代速度。能在CPU和GPU之间无缝切换，适用于不同的应用环境。当然，我们强烈推荐GPU环境。

首先这个思维导图是对Keras中文文档的整体概述，也可以叫做目录：



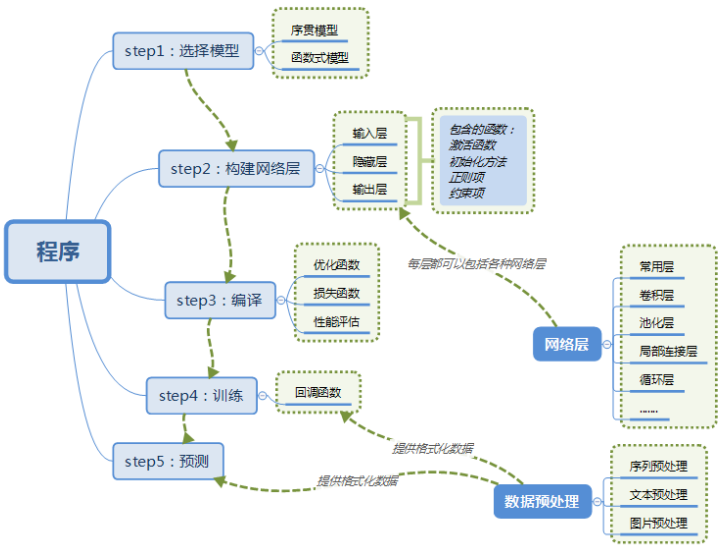
上面从上面导图我们可以直观的看到Keras官网文档主要分为五个方面写：模型，后端，网络层，网络配置，数据预处理。模型分为序贯模型和函数式模型，我们之前学习过就不赘述了；下面利用三个思维导图展示一下网络配置，网络层，数据预处理。（原图地址：https://blog.csdn.net/sinat_26917383/article/details/72857454?locationNum=1&fps=1）


keras-常见网络层
常用层
Dense层，全连接层
Activation层，激活层
Dropout层 Dropout层用于防止过拟合
Flatten层 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。
Reshape层 Reshape层用来将输入shape转换为特定的shape
Permute层 RGB->GBR，这样的维度重排
RepeatVector层 从(,32)到(3,32)，复制n份
Lambda层 对上一层的输出施以任何Theano/TensorFlow函数
ActivityRegularizer层 经过本层的数据不会有任何变化，但会基于其激活值更新损失函数值
Masking层 使用给定的值对输入的序列信号进行“屏蔽”，用以定位需要跳过的时间步
卷积层
Conv1D层 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。
(10,128)代表一个长为10的序列，序列中每个信号为128向量
(None, 128)代表变长的128维向量序列
Conv2D层 图像的空域卷积。
SeparableConv2D层 深度方向上的可分离卷积。
Conv2DTranspose层 转置的卷积操作（反卷积）
Conv3D层 三维卷积对三维的输入进行滑动窗卷积
Cropping1D层 时间轴（axis1）上对1D输入（即时间序列）进行裁剪
Cropping2D层 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
Cropping3D层 对2D输入（图像）进行裁剪
UpSampling1D层 在时间轴上，将每个时间步重复length次
UpSampling2D层 数据的行和列分别重复size[0]和size[1]次
UpSampling3D层 数据的三个维度上分别重复size[0]、size[1]和ize[2]次
ZeroPadding1D层 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
ZeroPadding2D层 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
ZeroPadding3D层 将数据的三个维度上填充0
池化层
MaxPooling1D层 时域1D信号进行最大值池化
MaxPooling2D层 空域信号施加最大值池化
MaxPooling3D层 3D信号（空域或时空域）施加最大值池化
AveragePooling1D层 时域1D信号进行平均值池化
AveragePooling2D层 空域信号施加平均值池化
AveragePooling3D层 3D信号（空域或时空域）施加平均值池化
GlobalMaxPooling1D层 对于时间信号的全局最大池化
GlobalAveragePooling1D层 时域信号施加全局平均值池化
GlobalMaxPooling2D层 空域信号施加全局最大值池化
GlobalAveragePooling2D层 空域信号施加全局平均值池化
局部连接层
LocallyConnected1D层 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。
LocallyConnected2D层
循环层
Recurrent层 类RNN
SimpleRNN层 全连接RNN网络，RNN的输出会被回馈到输入
GRU层 门限循环单元
LSTM层
嵌入层
Embedding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]]
融合层
融合两个层或两个张量的层对象和方法
Add 接收一个列表的同shape张量，并返回它们的和
Multiply，逐元素均值；Maximum，逐元素最大值
Dot，两个tensor中样本的张量乘积；
激活层
LeakyReLU层 Rectified Linear Unit，ReLU
PReLU层 参数化的ReLU（Parametric ReLU）
ELU层 指数线性单元（Exponential Linera Unit）
ThresholdedReLU层 门限的ReLU
规范化BN层
BatchNormalization层 每个batch上将前一层的激活值重新规范化
噪声层Noise
GaussianNoise层 为数据施加0均值，标准差为stddev的加性高斯噪声。
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
起正则化作用的层
GaussianDropout层 施加以1为均值，标准差为sqrt(rate/(1-rate)的乘性高斯噪声
起正则化作用的层
包装器Wrapper
TimeDistributed包装器 把一个层应用到输入的每一个时间步上
Bidirectional包装器 双向RNN包装器
http://blog.csdn.net/sinat_26917383

注意：回调函数 callbacks应该是Keras的精髓。。

3，一个简单的Keras训练模型过程

使用Keras训练模型的步骤图示如下：



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

3.1，选择模型

Sequential顺序模型如下所示：

```
from keras.models import Sequential

model = Sequential()
```

```
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

3.2，构建网络层

可以简单地使用.add()来堆叠模型：

```
from keras.layers import Dense

model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

3.3，编译

在完成了模型的构建后，可以使用.compile()来配置学习过程。编译模型时必须指明损失函数和优化器，如果有需要的话也可以自己定制损失函数。

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

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

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(lr=0.01, momentum=0.9, nesterov=True))
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

3.4 训练

现在我们可以批量的在训练数据上进行迭代了：

```
# x_train 和 y_train 是 Numpy 数组 -- 就像在 Scikit-Learn API 中一样。
model.fit(x_train, y_train, epochs=5, batch_size=32)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

或者，你可以手动地将批次的数据提供给模型：

```
model.train_on_batch(x_batch, y_batch)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

只需一行代码就能评估模型性能：

```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

3.5 预测

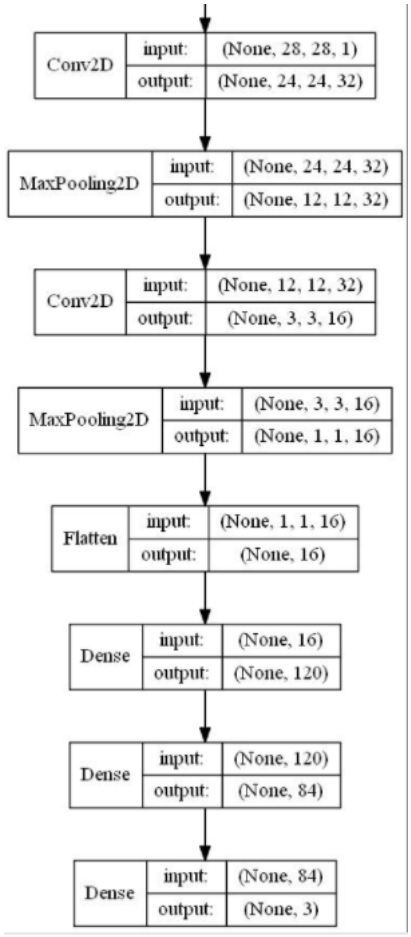
或者对新的数据生成预测：

```
classes = model.predict(x_test, batch_size=128)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

构建一个问答系统，一个图形分类模型，一个神经图灵机，或者其他的任何模型，就是这么的快。我们利用一个小的代码展示了一个Keras完整的训练过程，后面就不再赘述了。

3.6 保存模型结构

如果想打印看一下模型结果图，比如下图：



则需要使用 plot_model()函数，代码如下：

```
from keras.utils.vis_utils import plot_model


# write model image
plot_model(model, to_file='lenet.jpg', show_shapes=True, show_layer_names=False)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

4，一些Keras中常见的问题

4.1 为什么训练误差比测试误差高很多?

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。

4.2 在Theano和TensorFlow中如何表示一组彩色图片的尺寸?

Keras提供了两套后端，Theano和TensorFlow，这是一件幸福的事，就像手里拿着面包，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。但是如果你想使用一个已有的网络，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。

Theano和TensorFlow在表示一组彩色图片的问题上有分歧，“th”模式，也就是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张形式（100， 3， 16， 32），Caffe采取的也是这种形式。第0个维度为样本维，代表样本的树木，第一个维度是通道维，代表颜色通道数。后面两个就是高和宽了。这张Theano 风格的数据组织方式，称为“channels_first”， 即通道维靠前。

而TensorFlow 的表达形式（100， 16， 32，3），即把通道维放在了最后，这张数据组织形式称为“channels_last”。

注意：卷积核与所使用的后端不匹配，不会报任何错误，因为他们的shape是完全一致的，没有办法能够检测出这种错误。所以在使用预训练模型的时候，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_call_kernels_in_model 对模型的所有卷积核进行转换。

4.3，模型的节点信息提取

```
# 节点信息提取
config = model.get_config()
 # 把model中的信息，solver.prototxt和train.prototxt信息提取出来
model = Model.from_config(config)
# 还回去
# or, for Sequential:
model = Sequential.from_config(config)
# 重构一个新的Model模型，用去其他训练，fine-tuning比较好用
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

4.4 模型概况查询（包括权重查询）

```
# 1、模型概括打印
model.summary()

# 2、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)

# 3、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)

# 4、权重获取
model.get_layer()       #依据层名或下标获得层对象
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。

# 查看model中Layer的信息
model.layers 查看layer信息
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

4.5 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])
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

可以参考 官网：回调函数

4.6 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)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

4.7 如何在keras中设定GPU使用的大小

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，但是在使用keras时候会出现总是占满GPU显存的情况，可以通过重设backend的GPU占用情况来进行调节。

```
import tensorflow as tf
from keras.backend.tensorflow_backend import set_session
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.3
set_session(tf.Session(config=config))
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

需要注意的是，虽然代码或配置层面设置了对显存占用百分比阈值，但在实际运行中如果达到了这个阈值，程序有需要的话还是会突破这个阈值。换而言之如果跑在一个大数据集上还是会用到更多的显存。以上的显存限制仅仅为了在跑小数据集时避免对显存的浪费而已。

4.8 如何更科学的模型训练与模型保存

```
filepath = 'model-ep{epoch:03d}-loss{loss:.3f}-val_loss{val_loss:.3f}.h5'
checkpoint = ModelCheckpoint(filepath, monitor='val_loss', verbose=1, save_best_only=True, mode='min')
# fit model
model.fit(x, y, epochs=20, verbose=2, callbacks=[checkpoint], validation_data=(x, y))
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

save_best_only打开之后，会如下：

```
ETA: 3s - loss: 0.5820Epoch 00017: val_loss did not improve
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

如果val_loss 提高了就会保存，没有提高就不会保存。

4.9，如何在keras中使用tensorboard

```
RUN = RUN + 1 if 'RUN' in locals() else 1   # locals() 函数会以字典类型返回当前位置的全部局部变量。

    LOG_DIR = model_save_path + '/training_logs/run{}'.format(RUN)
    LOG_FILE_PATH = LOG_DIR + '/checkpoint-{epoch:02d}-{val_loss:.4f}.hdf5'   # 模型Log文件以及.h5模型文件存放地址

    tensorboard = TensorBoard(log_dir=LOG_DIR, write_images=True)
    checkpoint = ModelCheckpoint(filepath=LOG_FILE_PATH, monitor='val_loss', verbose=1, save_best_only=True)
    early_stopping = EarlyStopping(monitor='val_loss', patience=5, verbose=1)

    history = model.fit_generator(generator=gen.generate(True), steps_per_epoch=int(gen.train_batches / 4),
                                  validation_data=gen.generate(False), validation_steps=int(gen.val_batches /
                                  epochs=EPOCHS, verbose=1, callbacks=[tensorboard, checkpoint, early_stopping]
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

都是在回调函数中起作用：

- EarlyStopping patience：当early
  （1）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。
  （2）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。
- 模型检查点ModelCheckpoint
  （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
  （2） 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。
  （3）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）
  （4）period：CheckPoint之间的间隔的epoch数
- 可视化tensorboard write_images: 是否将模型权重以图片的形式可视化

4.10 模型概况查询（包括权重查询）

```
# 1、模型概括打印
model.summary()

# 2、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)

# 3、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)

# 4、权重获取
model.get_layer()       #依据层名或下标获得层对象
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。

```

```
# 查看model中Layer的信息
model.layers
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

4.11 二分类和多分类编译模型的参数设置

二分类编译模型的参数与多分类设置还是有区别的，具体如下：

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

# 多分类
model.compile(loss='categorical_crossentropy',                    # matt，多分类，不是binary_crossentropy
              optimizer='rmsprop',
              metrics=['accuracy'])
# 优化器rmsprop：除学习率可调整外，建议保持优化器的其他默认参数不变
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

4.12 Keras卷积补零相关的 border_mode 的选择以及 padding的操作

我们先来看看 Keras卷积操作中 border_mode 的实现：

```
# apply a 3x3 convolution with 64 output filters on a 256x256 image:
model = Sequential()
model.add(Convolution2D(64, 3, 3, border_mode='same', input_shape=(3, 256, 256))) # now model.output_shape ==

>> definition in keras:
def conv_output_length(input_length, filter_size, border_mode, stride):
    if input_length is None:
        return None
    assert border_mode in {'same', 'valid'}
    if border_mode == 'same':
        output_length = input_length
    elif border_mode == 'valid':
        output_length = input_length - filter_size + 1
    return (output_length + stride - 1) // stride
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

总结：如果卷积的方式选择为 same，那么卷积操作的输入和输出尺寸会保持一致。如果选择方式为 valid，那么卷积过后，尺寸会变小。

而TensorFlow中padding有两种方式，其中SAME是填充零，而VALID是不做填充。

（具体区别见：https://www.cnblogs.com/bugxch/p/14190955.html）

5，Keras中常用数据库Datasets

5.1 CIFAR10 小图片分类数据集

该数据库具有 50000个32*32 的彩色图片作为训练集，10000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()
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

返回值是两个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

5.2 CIFAR100 小图片分类数据集

该数据库具有 50000个32*32 的彩色图片作为训练集，10000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(lebel_mode='fine')
```

```
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

参数 label_model：为 fine 或 coarse，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。

返回值是两个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

5.3 MNIST 手写数字识别

该数据库具有 60000个28*28 的灰度手写数字图片作为训练集，10000个图片作为测试集

使用方法：

```
from keras.datasets import mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()<div class="open_grepper_editor" title="Edit & Save To (
```

参数 path：如果你本机上已经有此数据集（位于'~/.keras/datasets/'+path），则载入，否则数据将下载到该目录下。

返回值是两个Tuple。

X_train和X_test 是形如 （nb_samples, 28, 28）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

数据库会被下载到 ~/.keras/datasets/'+path

5.4 Boston 房屋价格回归数据库

该数据库由StatLib库取得，由CMU维护，每个样本都是 1970s晚期波士顿郊区的不同位置，每条数据含有13个属性，目标值是该位置房子的房价中位数（千 dollar）。

使用方法：

```
from keras.datasets import boston_housing

(X_train, y_train), (X_test, y_test) = boston_housing.load_data()<div class="open_grepper_editor" title="Edit &
```

参数 path：如果你本机上已经有此数据集（位于'~/.keras/datasets/'+path），则载入，否则数据将下载到该目录下。

参数 seed：随机数种子

参数 test_split：分割测试集的比例

返回值是两个Tuple。

X_train和X_test　　Y_train和Y_test

数据库会被下载到 ~/.keras/datasets/'+path

5.5 IMDB 影评倾向分类

本数据库含有来自 IMDB 的 25000 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所编码的词为数据中第三常出现的词。这样的组织方式使得用户可以快速完成诸如“只考虑最常出现的10000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.npz', num_words=None, skip_top=0,
    maxlen=None, seed=113,
    start_char=1, oov_char=2, index_from=3, )
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

参数path：如果你本机上已经有此数据集（位于'~/.keras/datasets/'+path），则载入，否则数据将下载到该目录下。

参数 nb_words：整数或None，要考虑的最常见的单词数，序列中任何出现频率更低的单词将会被编码为 oov_char 的值。

参数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 oov_char的值。

参数maxlen：整数，最大序列长度，任何长度大于此值的序列将会被截断。

参数 seed：整数，用于数据重排的随机数种子

参数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

参数oov_char：整数，因 nb_words或 skip_top 限制而 cut 掉的单词将被该字符代替

参数index_form：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

返回值是两个Tuple。

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_word-1.如果指定了 maxlen，则序列的最大可能长度为 maxlen

y_train和y_test 序列的标签，是一个二值 list

5.6 路透社新闻主题分类

该数据库包含来自路透社的11228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.npz', num_words=None, skip_top=0,
    maxlen=None, test_split=0.2, seed=113,
    start_char=1, oov_char=2, index_from=3,
)<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

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的下标：

```
word_index = reuters.get_word_index(path='reuters_word_index.json')
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

上面的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'] 的值可能是1234.

参数path：如果你在本机上有此数据集（位于 ~/.keras/datasets/'+path），则载入。否则数据将下载到该目录下

5.7 多分类标签指定Keras格式

数据集的载入上面都说了，而下面要强调的是Keras对多分类的标签需要一种固定格式，所以需要按照以下的方式进行转换，num_classes 为分类数量，假设此时有五类：

```
y_train = keras.utils.to_categorical(y_train, num_classes)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

最终输出的格式应该是（100， 5）

to_categorical 函数如下：

```
to_categorical(y,  num_classes = None)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

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。其中y表示类别向量，num_classes表示总共类别数。

6，延伸

6.1 fine-tuning 时如何加载 No_top 的权重

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)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

例如，假设元模型为：

```
    model = Sequential()
    model.add(Dense(2, input_dim=3, name="dense_1"))
    model.add(Dense(3, name="dense_2"))
    ...
    model.save_weights(fname)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

新模型如下：

```
# 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)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

6.2 应对不均衡样本的情况

使用 class_weight，sample_weight

两者的区别为

1. class_weight 主要针对的时数据不均衡问题，比如：异常检测的二项分类问题，异常数据仅占 1%，正常数据占 99 %；此时就要设置不同类对 loss 的影响。
2. sample_weight 主要解决的时样本质量不同的问题，比如前 1000 个样本的可信度，那么他的权重就要高，后 1000个样本可能有错，不可信，那么权重就要调低。

class-weight 的使用：

```
cw = {0: 1, 1: 50}
model.fit(x_train, y_train,batch_size=batch_size,epochs=epochs,verbose=1,callbacks=cbks,
validation_data=(x_test, y_test), shuffle=True,class_weight=cw)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

sample_weight 的使用：

```
from sklearn.utils import class_weight

list_classes = ["toxic", "severe_toxic", "obscene", "threat", "insult", "identity_hate"]
y = train[list_classes].values
sample_weights = class_weight.compute_sample_weight('balanced', y)

model.fit(X_t, y, batch_size=batch_size, epochs=epochs,validation_split=0.1,sample_weight=sample_weights, callba
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

来源：https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/discussion/46673

6.3 h5模型转pb模型，h5模型转 tflite模型

这里记录一下关于自己的h5模型转pb模型的代码，注意：**这里有基于Keras写的自定义损失函数如何保存到模型中。**

首先，我们看一下，如何将自定义损失函数保存到模型中：

```
from keras.models import load_model,

def contrastive_loss(y_true, y_pred):
    '''Contrastive loss from Hadsell-et-al.'06
        http://yann.lecun.com/exdb/publis/pdf/hadsell-chopra-lecun-06.pdf
        '''
    margin = 1
    square_pred = K.square(y_pred)
    margin_square = K.square(K.maximum(margin - y_pred, 0))
    return K.mean(y_true * square_pred + (1 - y_true) * margin_square)

h5_file = 'sample_model.h5'

h5_model = model.save(h5_file, custom_objects={'contrastive_loss': contrastive_loss})
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

保存为h5模型后，我们转pb模型（直接上代码）：

```
from keras.models import load_model
from tensorflow.python.framework import graph_util
from keras import backend as K
import tensorflow as tf
import os


def h5_to_pb(h5_file, output_dir, model_name, out_prefix="output_"):
    h5_model = load_model(h5_file, custom_objects={'contrastive_loss': contrastive_loss})
    out_nodes = []
    for i in range(len(h5_model.outputs)):
        out_nodes.append(out_prefix + str(i + 1))
        # print(out_nodes)  # ['output_1']
        tf.identity(h5_model.output[i], out_prefix + str(i + 1))
    sess = K.get_session()
    init_graph = sess.graph.as_graph_def()
    main_graph = graph_util.convert_variables_to_constants(sess, init_graph, out_nodes)
```

```
    with tf.gfile.GFile(os.path.join(output_dir, model_name), "wb") as filemodel:
        filemodel.write(main_graph.SerializeToString())
    print("pb model: ", {os.path.join(output_dir, model_name)})
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

这样转pb 是没有任何问题的，over。

我们用Keras训练模型后，通常保存的模型格式类型为 hdf5格式，也就是 .h5文件。但是如果如果我们想要移植到移动端，特别是基于TensorFlow支持的移动端，那就需要转化为tflite格式。

在TensorFlow高版本中支持通过命令行方式进行转换，如下：

```
tflite_convert --output_file=/my_model.tflite --keras_model_file=/my_model.h5
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

如果在程序中转换，加上自定义的损失函数，则代码如下：

```
from tensorflow import lite

def h5_to_tflite(h5_file, tflite_file):
    converter = lite.TFLiteConverter.from_keras_model_file(h5_file,
                                                           custom_objects={'contrastive_loss': contrastive_loss})
    tflite_model = converter.convert()
    with open(tflite_file, 'wb') as f:
        f.write(tflite_model)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

但是会出现问题，就是报错，转不了。。。。原因还未找到，如果不加损失函数，则没有问题，代码可以用。

还有一个 pb转 tflite的代码，我没有尝试。。。但是因为小改就可以用。

所有的代码如下：

```
import tensorflow as tf
# print(tensorflow.__version__)  # 1.14.0
from keras.models import load_model
from tensorflow.python.framework import graph_util
from tensorflow import lite
from keras import backend as K
import os


def h5_to_pb(h5_file, output_dir, model_name, out_prefix="output_"):
    h5_model = load_model(h5_file, custom_objects={'contrastive_loss': contrastive_loss})
    print(h5_model.input)
    # [<tf.Tensor 'input_2:0' shape=(?, 80, 80) dtype=float32>, <tf.Tensor 'input_3:0' shape=(?, 80, 80) dtype=
    print(h5_model.output)  # [<tf.Tensor 'lambda_1/Sqrt:0' shape=(?, 1) dtype=float32>]
    print(len(h5_model.outputs))  # 1
    out_nodes = []
    for i in range(len(h5_model.outputs)):
        out_nodes.append(out_prefix + str(i + 1))
        # print(out_nodes)  # ['output_1']
        tf.identity(h5_model.output[i], out_prefix + str(i + 1))
    sess = K.get_session()
    init_graph = sess.graph.as_graph_def()
    main_graph = graph_util.convert_variables_to_constants(sess, init_graph, out_nodes)
    with tf.gfile.GFile(os.path.join(output_dir, model_name), "wb") as filemodel:
        filemodel.write(main_graph.SerializeToString())
    print("pb model: ", {os.path.join(output_dir, model_name)})


def pb_to_tflite(pb_file, tflite_file):
    inputs = ["input_1"]  # 模型文件的输入节点名称
    classes = ["output_1"]  # 模型文件的输出节点名称
    converter = tf.lite.TocoConverter.from_frozen_graph(pb_file, inputs, classes)
    tflite_model = converter.convert()
    with open(tflite_file, "wb") as f:
        f.write(tflite_model)


def contrastive_loss(y_true, y_pred):
    '''Contrastive loss from Hadsell-et-al.'06
        http://yann.lecun.com/exdb/publis/pdf/hadsell-chopra-lecun-06.pdf
        '''
    margin = 1
```

```
    square_pred = K.square(y_pred)
    margin_square = K.square(K.maximum(margin - y_pred, 0))
    return K.mean(y_true * square_pred + (1 - y_true) * margin_square)


def h5_to_tflite(h5_file, tflite_file):
    converter = lite.TFLiteConverter.from_keras_model_file(h5_file,
                                                           custom_objects={'contrastive_loss': contrastive_loss
    tflite_model = converter.convert()
    with open(tflite_file, 'wb') as f:
        f.write(tflite_model)


if __name__ == '__main__':
    h5_file = 'screw_10.h5'
    tflite_file = 'screw_10.tflite'
    pb_file = 'screw_10.pb'
    # h5_to_tflite(h5_file, tflite_file)
    h5_to_pb(h5_file=h5_file, model_name=pb_file, output_dir='', )
    # pb_to_tflite(pb_file, tflite_file)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

保存为pb可以用在安卓端，这里使用pb来预测，和h5预测稍有不同，这里写的是使用pb来预测孪生网络的代码，和上面保持一致哈（其他的照着改就行）：

```
import tensorflow as tf
from tensorflow.python.platform import gfile
import cv2


def predict_pb(pb_model_path, image_path1, image_path2, target_size):
    sess = tf.Session()
    with gfile.FastGFile(pb_model_path, 'rb') as f:
        graph_def = tf.compat.v1.GraphDef()
        graph_def.ParseFromString(f.read())
        sess.graph.as_default()
        tf.import_graph_def(graph_def, name='')
    # 输入  这里有两个输入
    input_x = sess.graph.get_tensor_by_name('input_2:0')
    input_y = sess.graph.get_tensor_by_name('input_3:0')
    # 输出
    op = sess.graph.get_tensor_by_name('lambda_1/Sqrt:0')

    image1 = cv2.imread(image_path1)
    image2 = cv2.imread(image_path2)
    # 灰度化，并调整尺寸
    image1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
    image1 = cv2.resize(image1, target_size)
    image2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
    image2 = cv2.resize(image2, target_size)
    data1 = np.array([image1], dtype='float') / 255.0
    data2 = np.array([image2], dtype='float') / 255.0
    y_pred = sess.run(op, {input_x: data1, input_y: data2})
    print(y_pred)
<div class="open_grepper_editor" title="Edit & Save To Grepper"></div>
```

参考文献：https://blog.csdn.net/sinat_26917383/article/details/72859145

不经一番彻骨寒 怎得梅花扑鼻香

标签: 深度学习常用算法及笔记

« 上一篇： 数据竞赛实战（5）——方圆之外
» 下一篇： 我的Keras使用总结（2）——构建图像分类模型（针对小数据集）

posted @ 2020-03-21 09:20 战争热诚 阅读(12753) 评论(2) 编辑 收藏 举报