将Keras作为tensorflow的精简接口

文章信息

本文地址：https://blog.keras.io/keras-as-a-simplified-interface-to-tensorflow-tutorial.html

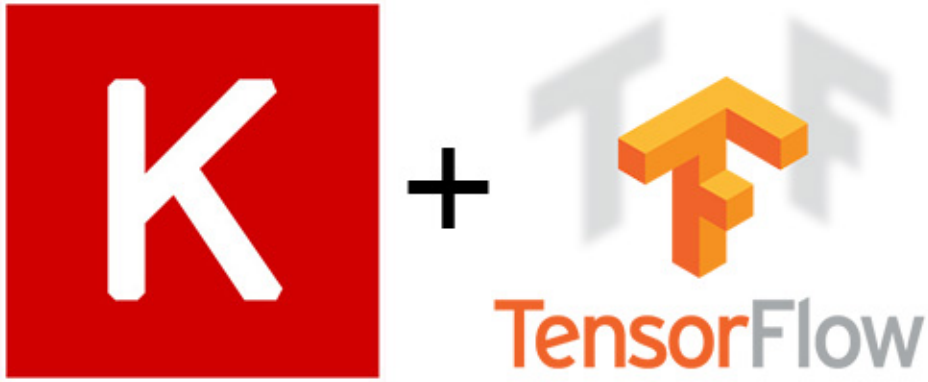
本文作者：Francois Chollet

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 = MyKerasModel(input_tensor)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