Keras as a simplified interface to TensorFlow: tutorial

A complete guide to using Keras as part of a TensorFlow workflow

Sun 24 April 2016
By Francois Chollet
In Tutorials.

If TensorFlow is your primary framework, and you are looking for a simple & high-level model definition interface to make your life easier, this tutorial is for you.

Keras layers and models are fully compatible with pure-TensorFlow tensors, and as a result, Keras makes a great model definition add-on for TensorFlow, and can even be used alongside other TensorFlow libraries. Let's see how.

Note that this tutorial assumes that you have configured Keras to use the TensorFlow backend (instead of Theano). Here are instructions on how to do this.

We will cover the following points:

I: Calling Keras layers on TensorFlow tensors

Let's start with a simple example: MNIST digits classification. We will build a TensorFlow digits classifier using a stack of Keras `Dense` layers (fully-connected layers).

We should start by creating a TensorFlow session and registering it with Keras. This means that Keras will use the session we registered to initialize all variables that it creates internally.

```
import tensorflow as tf
sess = tf.Session()

from keras import backend as K
K.set_session(sess)
```

Now let's get started with our MNIST model. We can start building a classifier exactly as you would do in TensorFlow:

```
# this placeholder will contain our input digits, as flat vectors
img = tf.placeholder(tf.float32, shape=(None, 784))
```

We can then use Keras layers to speed up the model definition process:

```
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
```

We define the placeholder for the labels, and the loss function we will use:

```
labels = tf.placeholder(tf.float32, shape=(None, 10))

from keras.objectives import categorical_crossentropy
loss = tf.reduce_mean(categorical_crossentropy(labels, preds))
```

Let's train the model with a TensorFlow optimizer:

```
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)

# Initialize all variables
init_op = tf.global_variables_initializer()
sess.run(init_op)

# Run training loop
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]})
```

We can now evaluate the model:

```
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})
```

In this case, we use Keras only as a syntactical shortcut to generate an op that maps some tensor(s) input to some tensor(s) output, and that's it. The optimization is done via a native TensorFlow optimizer rather than a Keras optimizer. We don't even use any Keras `Model` at all!

A note on the relative performance of native TensorFlow optimizers and Keras optimizers: there are slight speed differences when optimizing a model "the Keras way" vs. with a TensorFlow optimizer. Somewhat counter-intuitively, Keras seems faster most of the time, by 5-10%. However these differences are small enough that it doesn't really matter, at the end of the day, whether you optimize your models via Keras optimizers or native TF optimizers.

Different behaviors during training and testing

Some Keras layers (e.g. `Dropout`, `BatchNormalization`) behave differently at training time and testing time. You can tell whether a layer uses the "learning phase" (train/test) by printing `layer.uses_learning_phase`, a boolean: `True` if the layer has a different behavior in training mode and test mode, `False` otherwise.

If your model includes such layers, then you need to specify the value of the learning phase as part of `feed_dict`, so that your model knows whether to apply dropout/etc or not.

The Keras learning phase (a scalar TensorFlow tensor) is accessible via the Keras backend:

```
from keras import backend as K
print K.learning_phase()
```

To make use of the learning phase, simply pass the value "1" (training mode) or "0" (test mode) to `feed_dict`:

```
# train mode
train_step.run(feed_dict={x: batch[0], labels: batch[1], K.learning_phase(): 1})
```

For instance, here's how to add `Dropout` layers to our previous MNIST example:

```
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})
```

Compatibility with name scopes, device scopes

Keras layers and models are fully compatible with TensorFlow name scopes. For instance, consider the following code snippet:

```
x = tf.placeholder(tf.float32, shape=(None, 20, 64))
with tf.name_scope('block1'):
    y = LSTM(32, name='mylstm')(x)
```

The weights of our LSTM layer will then be named `block1/mylstm_W_i`, `block1/mylstm_U_i`, etc...

Similarly, device scopes work as you would expect:

```
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
```

Compatibility with graph scopes

Any Keras layer or model that you define inside a TensorFlow graph scope will have all of its variables and operations created as part of the specified graph. For instance, the following works as you would expect:

```
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
```

Compatibility with variable scopes

Variable sharing should be done via calling a same Keras layer (or model) instance multiple times, NOT via TensorFlow variable scopes. A TensorFlow variable scope will have no effect on a Keras layer or model. For more information about weight sharing with Keras, please see the "weight sharing" section in the functional API guide.

To summarize quickly how weight sharing works in Keras: by reusing the same layer instance or model instance, you are sharing its weights. Here's a simple example:

```
# 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)
```

Collecting trainable weights and state updates

Some Keras layers (stateful RNNs and `BatchNormalization` layers) have internal updates that need to be run as part of each training step. There are stored as a list of tensor tuples, `layer.updates`. You should generate `assign` ops for those, to be run at each training step. Here's an example:

```
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))
```

Note that if you are using a Keras model (`Model` instance or `Sequential` instance), `model.udpates` behaves in the same way (and collects the updates of all underlying layers in the model).

In addition, in case you need to explicitly collect a layer's trainable weights, you can do so via `layer.trainable_weights` (or `model.trainable_weights`), a list of TensorFlow `Variable`instances:

```
from keras.layers import Dense

layer = Dense(32)(x)  # instantiate and call a layer
print layer.trainable_weights  # list of TensorFlow Variables
```

Knowing this allows you to implement your own training routine based on a TensorFlow optimizer.

II: Using Keras models with TensorFlow

Converting a Keras `Sequential` model for use in a TensorFlow workflow

You have found a Keras `Sequential` model that you want to reuse in your TensorFlow project (consider, for instance, this VGG16 image classifier with pre-trained weights). How to proceed?

First of all, note that if your pre-trained weights include convolutions (layers `Convolution2D` or `Convolution1D`) that were trained with **Theano**, you need to flip the convolution kernels when loading the weights. This is due

Theano and TensorFlow implementing convolution in different ways (TensorFlow actually implements correlation, much like Caffe). Here's a short guide on what you need to do in this case.

Let's say that you are starting from the following Keras model, and that you want to modify so that it takes as input a specific TensorFlow tensor, `my_input_tensor`. This input tensor could be a data feeder op, for instance, or the output of a previous TensorFlow model.

```
# this is our initial Keras model
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=784))
model.add(Dense(10, activation='softmax'))
```

You just need to use `keras.layers.InputLayer` to start building your Sequential model on top of a custom TensorFlow placeholder, then build the rest of the model on top:

```
from keras.layers import InputLayer

# this is our modified Keras model
model = Sequential()
model.add(InputLayer(input_tensor=custom_input_tensor,
                     input_shape=(None, 784)))

# build the rest of the model as before
model.add(Dense(32, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

At this stage, you can call `model.load_weights(weights_file)` to load your pre-trained weights.

Then you will probably want to collect the `Sequential` model's output tensor:

```
output_tensor = model.output
```

You can now add new TensorFlow ops on top of `output_tensor`, etc.

Calling a Keras model on a TensorFlow tensor

A Keras model acts the same as a layer, and thus can be called on TensorFlow tensors:

```
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)
```

Note: by calling a Keras model, your are reusing both its architecture and its weights. When you are calling a model on a tensor, you are creating new TF ops on top of the input tensor, and these ops are reusing the TF `Variable` instances already present in the model.

III: Multi-GPU and distributed training

Assigning part of a Keras model to different GPUs

TensorFlow device scopes are fully compatible with Keras layers and models, hence you can use them to assign specific parts of a graph to different GPUs. Here's a simple example:

```
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
```

Note that the variables created by the LSTM layers will not live on GPU: all TensorFlow variables always live on CPU independently from the device scope where they were created. TensorFlow handles device-to-device variable transfer behind the scenes.

If you want to train multiple replicas of a same model on different GPUs, while sharing the same weights across the different replicas, you should first instantiate your model (or layers) under one device scope, then call the same model instance multiple times in different GPU device scopes, such as:

```
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})
```

Distributed training

You can trivially make use of TensorFlow distributed training by registering with Keras a TF session linked to a cluster:

```
server = tf.train.Server.create_local_server()
sess = tf.Session(server.target)

from keras import backend as K
K.set_session(sess)
```

For more information about using TensorFlow in a distributed setting, see this tutorial.

IV: Exporting a model with TensorFlow-serving

TensorFlow Serving is a library for serving TensorFlow models in a production setting, developed by Google.

Any Keras model can be exported with TensorFlow-serving (as long as it only has one input and one output, which is a limitation of TF-serving), whether or not it was training as part of a TensorFlow workflow. In fact you could even train your Keras model with Theano then switch to the TensorFlow Keras backend and export your model.

Here's how it works.

If your graph makes use of the Keras learning phase (different behavior at training time and test time), the very first thing to do before exporting your model is to hard-code the value of the learning phase (as 0, presumably, i.e. test mode) into your graph. This is done by 1) registering a constant learning phase with the Keras backend, and 2) re-building your model afterwards.

Here are these two simple steps in action:

```
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)
```

We can now use TensorFlow-serving to export the model, following the instructions found in the official tutorial:

```
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)
```

Want to see a new topic covered in this guide? Reach out on Twitter.