TensorFlow program via XML

Summary

This article demonstrates separation of the Neural Network problem specification and its solution code. In this approach, problem dataset and its Neural network are specified in a PMML like XML file. Then it is used to populate the TensorFlow graph, which, in turn run to get the results.

Iris dataset is used as a data source in this approach. With suitable enhancements, other data sources, even different Neural Network types and other libraries can also be incorporated in it.

Introduction

In the initial part of a typical Deep Learning program, such as TensorFlow, you need to define the architecture (flow diagram) of the neural network you wish to solve. Here, you are constructing what is called as the TensorFlow graph. Its nodes are the operations and edges are the lines through which data flows, in the form of tensors. Actual input data is not supplied during this construction but only placeholders are created for them. Towards end, this graph is executed in TensorFlow session's run method. At this place, the actual input data is supplied in the form of feeds. Reasons for the separation of building symbolic graph first and then executing it later, are many. For example, avoiding multiple data transfers between python and backend C++ at each step, leveraging hardware (CPU/GPU) and of course, parallelization once all the data and parameters are known.

Major portion of a typical TensorFlow program goes into specifying the graph. One must know the TensorFlow data types, the sequence of operations as well as the APIs. Obvious question that comes to mind is: can this process be made more user friendly? Graphical User Interface would be the easiest way, but may not be the quickest. How about specifying the neural network problem in a simple XML file? Following article describes this XML way. It elaborates use of PMML like specification to define the problem, how TensorFlow graph is populated (and run) using it.

Neural Network Problem Specification

Iris dataset classification problem is specified in the XML format as below (data/iris_to_tf.pmml):

```
<?xml version='1.0' encoding='utf-8'?>
<PMML version="4.2.1" xmlns="http://www.dmg.org/PMML-4_2">
  <Header copyright="Yogesh Kulkarni" description="Input Model for Tensorflow">
    <Timestamp>2017-05-29 09:33:59.170790</Timestamp>
  </Header>
  <MiningBuildTask>
    <Extension>&lt;?xml version="1.0" encoding="iso-8859-15" ?&gt;
      <DataMining>
        <MiningTask>
          <MiningData name="training_set" filename="data/iris_training.csv"/>
          <MiningData name="test_set" filename="data/iris_test.csv"/>
        </MiningTask>
      </DataMining>
    </Extension>
  </MiningBuildTask>
  <NeuralNetwork learning_rate="0.1" num_epochs="10000" regularization_strength ="0.1"  hidden_layer_size="10">
    <NeuralLayer function="relu"/>
    <NeuralLayer function="softmax_cross_entropy_with_logits"/>
  </NeuralNetwork>
</PMML>
```

Initial few lines of the .pmml file specify version, header information, etc. Data sources are specified in the MiningBuildTask block. It mentions training and testing data set files.

Next block "NeuralNetwork" specifies the 2-layer simple neural network. The hyper parameters such as number of epochs, learning rate, etc are specified. Inside this block, list of NeuralLayers are specified. Each layer specifies the activation/loss function. There are far more parameters than the ones shown here and this program can surely be generalized extensively to take care of them as well. It can also be extended to cater to different data sources, neural network types, pre-processing methods, output specifications, etc. PMML standard, deals with many of those aspects and thus making current program fully compliant can be taken up later.

TensorFlow Graph population

The Neural network problem specification are parsed using ElementTree library. The parser is based on the implementation seen in Vaclav's Titanium library. It primarily dealt with reading Neural network blocks. It has been enhanced to include MiningBuildTasks for specified the data sources.

For the example pmml specifications shown above, the following text box shows the data read as pmml object. These are the only parameters used to populate the TensorFlow graph.

```
Data Sources: {'training_set': 'data/iris_training.csv', 'test_set': 'data/iris_test.csv'}
Learning rate: 0.1
Num Epochs: 10000
Regularization Strength: 0.1
Num Layers: 2
Functions: ['relu', 'softmax_cross_entropy_with_logits']
```

Typical TensorFlow graph population steps are mentioned below. It has been developed by refering to a typical TensorFlow program (refer original StackOverlfow post):

- Read the data. In the iris example, it is reading the csv files. Outcome or class labels are converted to One Hot encoding. Code shown below is used for both training and test data. Hardcoded parameter like data types, target column, can also be specified in the pmml file (but not done so in this example.

```
training_set = tf.contrib.learn.datasets.base.load_csv_with_header(filename=training_filename,
                                                   target_dtype=np.int,
                                                   features_dtype=np.float32,
                                                   target_column=-1)

self.training_features    = training_set.data.astype(np.float32)
self.training_labels      = training_set.target
self.training_labels_1hot = self.convert_to_one_hot(self.training_labels)
```

- Populate place holders for x (features) and y (labels or classes).
- For each layer (2 in this example), variables for Weights and Biases are allocated and initialized.
- Hidden layer size (h1_size) specified in the pmml is used in the allocation as:
  - W1 (num_features x h1_size)
  - b1 (h1_size)
  - W2 (h1_size x num_classes)
  - B2 (num_classes)

- Activation function for Layer 1 and loss function for Layer 2 (i.e. output layer) are used from pmml specification, as below:

```
H1 = tf.matmul(x, W1) + b1
fn_1 = self.function_dict[self.pmml._functions[0]]
H1 = fn_1(H1)

y_hat = tf.matmul(H1, W2) + b2
fn_2 = self.function_dict[self.pmml._functions[1]]

J = tf.reduce_mean(fn_2(y_hat, y_ph) + \
      regularization_strength * tf.nn.l2_loss(W1) + \
      regularization_strength * tf.nn.l2_loss(W2))
```

- Training step is GradientDescentOptimizer for the loss function "J".
- Session is run in the specified number of epochs as below:

```
for i in range(num_epochs):
  op, J_result = sess.run([train_step, J], feed_dict={x:self.data.training_features,
                                                      y:  self.data.training_labels_1hot})
```

- Once the run is over, a model is built by querying W1, b1, W2, b2 from the session object. This model can be further used for predicting unseen data.
- Results of the run show accuracy of about 96%.
- The driver program can be as simple as shown below:

```
from nn_pmml_reader import NN_PMML_Reader
from tensorflow_pmml import Tensorflow_PMML

pmml = NN_PMML_Reader()
pmml.read_pmml('data/iris_for_tf.pmml')

classifier = Tensorflow_PMML(pmml)
classifier.loadData()
model = classifier.trainModel()

model.print() # Do something with 'model' if needed.
```

Implementation of the parser, TensorFlow program, along with the iris pmml file is at GitHub.

End Note

The Neural network problem specification in XML provides an easy-to-specify approach for describing the TensorFlow graph. For any change in problem parameters, only the XML needs to be changed. The core TensorFlow program remains untouched. Another advantage is that, as the specification use commonly used parameter for the neural network types, options, etc. their Internal TensorFlow equivalents, can be adjusted based on the current version. So, if there is any change in TensorFlow APIs over versions, this program takes care of the switch. This, thus, avoids problems caused due to depreciations (seen "initialize variables" depreciation warnings?).

This approach can be extended further to be used with other Deep Learning libraries as well.