Karlsruhe Institute of Technology

High Performance Humanoid Technologies

Introducing TensorFlow to H2T

Jonas Rothfuss, Fabio Ferreira | November 6, 2017

HIGH PERFORMANCE HUMANOID TECHNOLOGIES (H2T)

Outline

Overview

TensorFlow[1] is a symbolic open-source software library for numerical computation using data flow graphs (Abadi et al. 2016)

- suitable for both research & production
- currently available for Python, C/C++ and Java
- embedded applications: Mobile TensorFlow (Raspberry Pi, Android, iOS)

[1] TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc.

Overview

- CPU support
- GPU support for Nvidia GPUs (requires CUDA and cuDNN)
- for installation see: [tensorflow manual]
- for workstations@H2T see: [H2T Deep Learning Wiki]

Preliminary

Best Practices

- use Python's *virtualenv* along with pip
- if your environment allows it: use *Docker* images (not at H2T)
- use *tcmalloc* (memory allocator for high concurrency situations) for a tremendously more efficient resource allocation during training

- for more Best Practices, see: H2T Deep Learning Wiki

TensorFlow's Symbolic Programming

Two distinctive phases during development:

symbolic programming paradigm

- **phase 1**: build a computation graph of operations
- **phase 2**: convert the graph into a function (compilation) and execute it in a session

(Bad) consequences:

- computation happens as the last step in the code
- debugging code is usually difficult
- native python statements must be provided in TensorFlow language
- special statements for typical control flow e.g. *tf.while_loop*

TensorFlow's Symbolic Programming

Two distinctive phases during development:

symbolic programming paradigm

- **phase 1**: build a computation graph of operations
- **phase 2**: convert the graph into a function (compilation) and execute it in a session

(Good) consequences :

- execution is efficient (memory, in-place computation)
- preprocessing and data loading is simply done by adding operations to graph
- distribute computation among different resources (multi-gpu, clusters)

TensorFlow's Symbolic Programming

Some TensorFlow terminology first before showing some example code:

Definition

Tensor: Is a typed (float32, int32, string) multi-dimensional array.

For example a mini-batch of 1-channel-images as a 2D-array of floating point numbers with dimensions [batch, width*height]:

```
x = tf.placeholder(tf.float32, [None, 784])
```

TensorFlow's Symbolic Programming

Karlsruhe Institute of Technology

Definition

Tensor: Is a typed (float32, int32, string) multi-dimensional array.

Some widely used types are: *tf.Variable:*

- initial values are required
- for parameters to learn (and save/restore)

tf.Placeholder:

- initial values are *not* required
- for the allocation of storage

TensorFlow’s Symbolic Programming

Karlsruhe Institute of Technology

Definition

Operation: A node in the computation graph. Takes zero or more Tensors as input, performs computations with them and produces zero or more Tensors.

for example assign a matrix multiplication operation/node to the graph:

```
a_1 = tf.matmul(x,W_1)+b_1
```

TensorFlow's Symbolic Programming

Definition

Session: For any computation, a graph must be launched in a Session. The Session places the graph onto CPUs or GPUs and provides methods to execute it. Methods executed in a Session return Tensors produced by ops as **numpy ndarray** objects.

Launch the (default) graph by initializing a session:

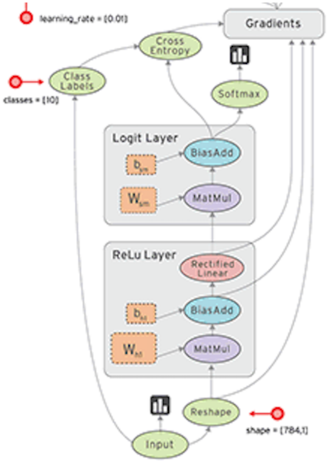
```
sess = tf.InteractiveSession()
```

Calling run() will execute all specified operations in the graph:

```
loss, _ = sess.run([cross_entropy, train_op], feed_dict={x: batch_x, y: batch_y})
```

TensorFlow's Symbolic Programming

An exemplary graph with input, neurons (weights, biases), softmax and cross entropy function:



Workflow of training a NN

Decide on how to feed the Data → Define the Network Structure → Define Operations for Execution → Execute Network in a Session

Try out the enclosed example code by running the script *start_tutorial* from the command line.

Data Pre-Processing

Our main advice for dealing with data is to invest a decent amount of time (25-30% of total project time) into pre-processing. Here are some questions you should ask (and potential advices in brackets):

- **Is the dataset large enough or will I need data augmentation?**
  ($\rightarrow$ plan in more time if it's too small, validate your augmentation by writing test methods for the functions)
- **How likely will I work with more than one dataset?**
  ($\rightarrow$ think about a uniform dataset pipeline, use tf.flags for differing between datasets)
- **Which use-cases (e.g. just training a NN, I/O pipeline for a robot demo) will I be confronted with?**
  ($\rightarrow$see next two slides)

Three Ways of Importing Data

Three main ways exist to reading data into TensorFlow:

- **Feeding**: Python code provides a sample from data every single iteration
- **Reading**: use an input pipeline for reading the data from files (typically .tfrecords, .csv or binary/encoded data), data fetching is incorporated into the computation graph
- **Preloaded Data**: load a (small) data set entirely into the memory by using constants or variables

See the [full documentation for Reading Data].

Three Ways of Importing Data

When to use what?

- use **Feeding**
  - to get familiar with TensorFlow
  - for live-demos
  - for tunneling data through third-party services/pipelines (e.g. ICE)
- use **Reading** if
  - your data set is too large to be stored in the memory directly
  - you need to efficiently train a NN
  - you want to benefit from native pipeline functions (e.g. automatic batch-creation/-handling, Threading & Queuing)
- use **Preloaded Data** if
  - your data set is small enough to be stored in the memory directly
  - you want efficiency in training an NN and
  - benefit from the the native pipeline (see *Reading*)

Importing Data with the Dataset API

Karlsruhe Institute of Technology

TF has recently (r1.2.) introduced a new Dataset API for dealing with large amounts of data, different file formats and transformations (e.g. batches) which also incorporates Threading & Queuing. If you want to use *Dataset*, principle steps are:

- **Decide on the abstraction level**: *tf.data.Dataset* is a sequence of elements and every element contains one or more Tensors (e.g. in an image pipeline every element is a single image)
- **Define the source of your data set**: to start an input pipeline, you must specify a source, e.g. by loading the Tensors from memory or TFRecords
- **Apply transformation**: once a source is defined, you can transform the Dataset into a new Dataset that suffices your requirements (e.g. creating a batch or using *Dataset.map()* to apply a function to each element)

Existing (old) Input Pipeline methods will be available until TF 2.0 (at least)

TensorFlow's TFRecords

TFRecords is a supported format for any type of data. Creating the records can sometimes be a hassle but once they exist you'll benefit from helpful native pipeline functions.

For generating TFrecords one ideally writes a (1st) routine that:

- reads-in the data (e.g. 3 channel image in numpy array)
- place the data into a protocol buffer *tf.Example* object
- serializes the protocol buffer into a string
- writes the string into a TFRecords file using the *tf.TFRecordWriter*

TensorFlow's TFRecords

On the other side, to use the generated TFRecords, one writes a (2nd) routine that:

- reads-in the records with *tf.TFRecordReader*
- decodes the data with *tf.parse_single_example* to receive the deserialized data as a tensor for a single sample
- uses the tensor as a template to create a batch for training *tf.train.batch*
- next, choose the size of the batch (how many of template tensors need to be in a batch) to receive a new tensor (representing the batch)
- provides the new tensor to the model at initialization (before *sess.run()*

Examples can be found here (Deep Episodic Memory) and here (TensorFlow doc).

Useful remarks: Save & Restore a Graph

Saving & restoring a graph is

- typically useful when a training should be saved, e.g. after 1000 training iterations / 1 hour
- or if you want your results to be reproducible

For saving & restoring examples: see here (Deep Episodic Memory) and here (TensorFlow doc).

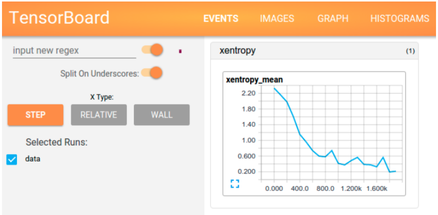
Fine-tuning a pre-trained model: this has touching points with the saver. If you need to exclude some variables from the graph for training, see this code.

Recommended: create a protocol of parameters (solver, output directory, dataset etc.) you specified for your trainings, e.g. as done here or directly use config files.

Useful remarks: Visualization

Visualization can help track down problems in your architecture/code but can also help doing sanity checks and help monitoring the training progress. Visualization is done with TensorBoard.

- Graph Visualization (can help finding structural errors when code is already to complex)
- Visualizing Learning (monitor your training progress)



You can access TensorBoard via ssh from different computers, see here

Useful remarks: Debugging

Due to symbolic programming, debugging TensorFlow programs is generally hard. To make things easier, the authors have introduced two helpful tools:

- TensorFlow debugger (tfdbg): debug-wrapper around *sess.run()* that allows a program to start with *–debug* flag → interactive debug mode in console
- newly introduced: TensorFlow Eager Execution, an interface for imperative programming style. Activate eager execution and execute TF operations immediately without executing a graph with *sess.run()*

Concluding

Things we considered most challenging during Deep Episodic Memory

Challenges

- setting-up the training pipeline with tfrecords
- keeping track of changes and decisions that affected the design of the model
- finding bugs (or assess the effect of code changes) in symbolic programming

References I

Martín Abadi et al. “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”. In: *CoRR* abs/1603.04467 (2016). arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467.