**TVMDBG**

TVM Debugger (TVMDBG) is a specialized debugger for TVM's computation graphs. It provides access to internal graph structures and tensor values at TVM runtime.

**Why TVMDBG**

In TVM's current computation-graph framework, almost all actual computation after graph construction happens in a single Python function, namely [t](https://www.tensorflow.org/api_docs/python/tf/Session" \l "run)vm.run. Basic Python debugging tools such as [pdb](https://docs.python.org/2/library/pdb.html) cannot be used to debug tvm.run, due to the fact that TVM's graph execution happens in the underlying C++ layer. C++ debugging tools such as [gdb](https://www.gnu.org/software/gdb/) are not ideal either, because of their inability to recognize and organize the stack frames and variables in a way relevant to TVM's operations, tensors and other graph constructs.

TVMDBG addresses these limitations. Among the features provided by TVMDBG, the following ones are designed to facilitate runtime debugging of TVM models:

* Easy access through session wrappers
* Inspection of runtime tensor values and node connections
* Conditional breaking after runs that generate tensors satisfying given predicates, which makes common debugging tasks such as tracing the origin of infinities and [NaNs](https://en.wikipedia.org/wiki/NaN) easier
* Association of nodes and tensors in graphs with Python source lines

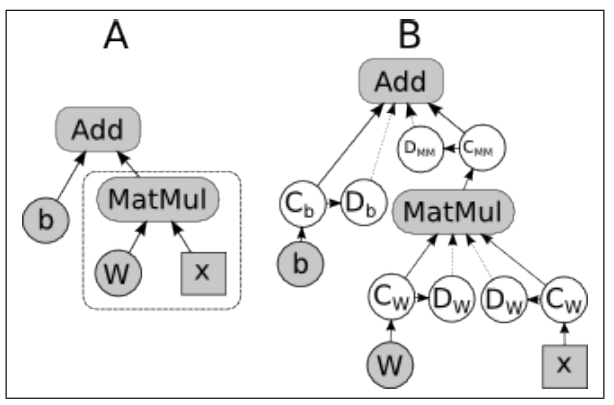
**How to use TVM?**

* TVMDBG command-line interface
* For programmatic use of the API of TVMDBG

This guide focuses on the command-line interface (CLI) of tvmdbg.

**Note:** The TVM debugger uses a curses-based text user interface. On Mac OS X, the **ncurses** library is required and can be installed with **brew install homebrew/dupes/ncurses**. On Windows, curses isn't as well supported, so a readline-based interface can be used with tfdbg by installing **pyreadline** with **pip**. If you use Anaconda3, you can install it with a command such as **"C:\Program Files\Anaconda3\Scripts\pip.exe" install pyreadline**. Unofficial Windows curses packages can be downloaded here, then subsequently installed using **pip install <your\_version>.whl**, however curses on Windows may not work as reliably as curses on Linux or Mac.

This tutorial demonstrates how to use the **tvmdbg** CLI to debug the appearance of [**nan**s](https://en.wikipedia.org/wiki/NaN) and [**inf**s](https://en.wikipedia.org/wiki/Infinity), a frequently-encountered type of bug in TVM model development.

****

**Section A.** An example dataflow graph in NNVM. Nodes Add and MatMul are computation nodes. W and b are Variables. x is an Placeholder node. The dashed box provides an example for tvmdbg NodeStepper’s stepping through a graph. Suppose the nodes in the dashed box have been executed in a previous continue call, a subsequent continue call on the Add node need not recompute those nodes, but can use the cached tensor for the MatMul node.

**Section B.** Debug nodes inserted by tvmdbg are shown as white circles. Each node in the original graph is watched by a pair of nodes, a Copy (C) node and a Debug (D) node that exports the value copy to a configurable location.

**Analyzer**

The Analyzer adds observability to the graph execution process. It makes the structure and intermediate state of the runtime graph visible.

**Design of the Analyzer.** The Analyzer inserts special-purpose debugging nodes into the runtime graph itself and lets those nodes export the data. This approach provides flexibility in expanding or changing the functionality of the data-exporting debug nodes and greater portability among different runtime environments. As Figure B shows, to watch a node’s output, a pair of Copy (C) and Debug (D) nodes are inserted between the output slot and its outgoing targets (recipients). The Copy node copies the watched tensor, to guard against updates to the tensor value by concurrent threads in a race condition. The debug ops support dumping the values of the watched tensors to the file system and sending them remotely through remote procedural calls (RPC). The file dumping mode is used for debugging on a single machine, where the total size of the intermediate tensors in a single run or across multiple runs can exceed the RAM of the machine. The RPC mode is used during distributed training, where a centralized debugging server receives the debug data and may store them to the file system for subsequent analysis. In addition to dumping the intermediate tensors, the Analyzer also exposes the runtime graphs, which generally differ from the graph set up by the client in a front-end language, e.g., Python. The Analyzer provides options to watch a subset of the nodes on the graph, e.g., by filtering node names with a given regular-expression pattern. The Analyzer also has a limited amount of built-in intelligence to analyze common model issues, such as highlighting the backward path in a training graph and determining the source node of bad numerical values.

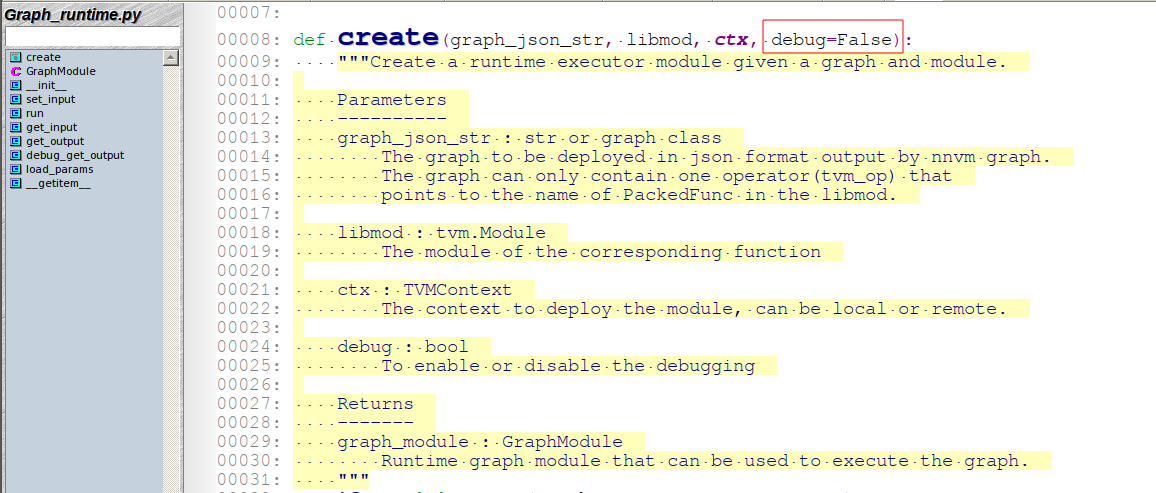
**NNVM Design**

**1. Disable the optimization being done by NNVM.**

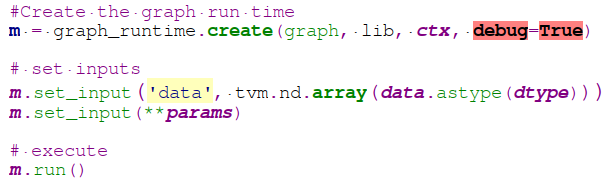
**2. Insert copy node and Debug Node for Compute node.**

**TVM Design**

**In tvm/contrib/graph\_runtime.py debug flag is added.**



To enable the debugging session, developer can set the below debug flag when graph runtime is created.



If debug is enabled, during graph\_runtime.create

graph\_runtime.create

Debug

enabled

Normal Flow

Initalize curses

framework

Load the graph

for curses

NO

YES

graph\_runtime

set\_input

Debug

enabled

Normal Flow

Load the params

or input data

for curses

NO

YES

**For Run, allocate memory from the and send..**

**Issues:**

1. Can dump only fused graph.

2. Layer information will be dispersed into multiple operators.

**Use Cases**

**White-box Testing of ML Models.** In an ML system under active development, feature development and code refactoring can sometimes lead to unforeseen changes in the structure and behavior of an ML model. Such changes can also arise as a result of changes in the underlying ML library itself. If left untested, these low-level changes can lead to subtle issues in production that are difficult to observe and debug. The above-described Analyzer module of tvmdbg makes it possible to make assertions about a model in unit tests.

Two types of assertions can be made based on tvmdbg’s Analyzer:

1) Structural assertions: The structure of a NNVM graph, the nodes and their attributes

2) Functional assertions: The intermediate tensor values in the graph under a given set of inputs. Structural and functional assertions should focus on the critical parts of the model, such as output of a neural-network layer or an embedding lookup result, and ignore noncritical parts, in order to avoid being sensitive to unimportant changes caused by refactoring or library changes.

**Debugging Problematic Numerical Values.** A type of frequently encountered problem in ML model training/inference is bad numerical values, e.g., **infinities and NaNs**, which arise due to various reasons such as numerical overflow and underflow, logarithm of and division by zero. In a large ML model with thousands of nodes, it can be hard to find the node at which this first emerged and started propagating through the graph. With tvmdbg, the user can specify a breaking predicate in the RunStepper to let runs break when any intermediate tensors in the model first show infinities or NaNs and drop into the Analyzer UI to identify the first-offending node. By examining the type of node and its inputs using the Analyzer UI, the user can obtain useful information about why these values occur, which often leads to a fix to the issue such as applying value clipping to the problematic node.

**Directory Structure:**

**NNVM**

├── amalgamation

├── cmake

├── dmlc-core

├── docs

├── examples

│ ├── benchmark

│ ├── sgx

│ └── tools

├── include

├── lib

├── make

├── python

│ ├── conda

│ ├── dist

│ └── nnvm

│ ├── compiler

│ ├── \_ctypes

│ ├── \_cy2

│ ├── \_cy3

│ ├── cython

│ ├── frontend

│ ├── testing

│ ├── top

│ └── tools

├── src

│ ├── c\_api

│ ├── compiler

│ ├── core

│ ├── pass

│ ├── tools

│ │ └── debug

│ └── top

│ ├── nn

│ ├── tensor

│ └── vision

│ └── yolo2

├── tests

│ ├── ci\_build

│ │ └── install

│ ├── cpp

│ ├── lint

│ ├── python

│ │ ├── compiler

│ │ ├── frontend

│ │ ├── unittest

│ │ └── tools

│ ├── scripts

│ └── travis

├── tutorials

└── web

**TVM**

├── apps

│ ├── android\_rpc

│ │ ├── app

│ │ │ └── src

│ │ │ └── main

│ │ │ └── java

│ │ │ └── ml

│ │ │ └── dmlc

│ │ │ └── tvm

│ │ │ └── tvmrpc

│ │ │ └── tools

│ │ ├── dev\_tools

│ │ └── tests

│ ├── extension

│ ├── howto\_deploy

│ ├── ios\_rpc

│ └── sgx

├── docs

├── include

├── jvm

├── make

├── python

│ ├── conda

│ └── tvm

│ ├── contrib

│ ├── exec

│ ├── \_ffi

│ └── tools

│ ├── debug

│ └── profiler

├── sgx

├── src

│ ├── api

│ ├── arithmetic

│ ├── codegen

│ ├── common

│ ├── contrib

│ ├── lang

│ ├── op

│ ├── pass

│ ├── runtime

│ ├── tools

│ │ ├── debug

│ │ └── profiler

│ └── schedule

├── tests

│ ├── ci\_build

│ ├── cpp

│ ├── lint

│ ├── python

│ ├── scripts

│ ├── travis

│ ├── verilog

│ ├── web

│ └── webgl

└── topi

├── include

└── python

├── tutorials

└── verilog