


Branch: master ▾

tensorflow-on-raspberry-pi / GUIDE.md




Find file

Copy path

 pbabbicola

Update Bazel edits for 9.0.0

24d5473 on Jan 18

3 contributors   

380 lines (270 sloc) 13.9 KB

Building TensorFlow for Raspberry Pi: a Step-By-Step Guide

[Back to readme](#)

What You Need

- Raspberry Pi 2 or 3 Model B
- An SD card running Raspbian with several GB of free space
 - An 8 GB card with a fresh install of Raspbian **does not** have enough space. A 16 GB SD card minimum is recommended.
 - These instructions may work on Linux distributions other than Raspbian
- Internet connection to the Raspberry Pi
- A USB memory drive that can be installed as swap memory (if it is a flash drive, make sure you don't care about the drive). Anything over 1 GB should be fine
- A fair amount of time

Overview

These instructions were crafted for a [Raspberry Pi 3 Model B](#) running a vanilla copy of Raspbian 8.0 (jessie). It appears to work on Raspberry Pi 2, but [there are some kinks that are being worked out](#). If these instructions work for different distributions, let me know! Updated (2017-09-11) to work with the latest (HEAD) version of tensorflow on Raspbian Stretch (Vanilla version september 2017) and Python 3.5.

Here's the basic plan: build a RPi-friendly version of [Bazel](#) and use it to build TensorFlow.

Contents

1. [Install basic dependencies](#)
2. [Install USB Memory as Swap](#)
3. [Build Bazel](#)
4. [Compiling TensorFlow](#)
5. [Cleaning Up](#)
6. [References](#)

The Build

1. Install basic dependencies

First, update apt-get to make sure it knows where to download everything.

```
sudo apt-get update
```

Next, install some base dependencies and tools we'll need later.

For Bazel:

```
sudo apt-get install pkg-config zip g++ zlib1g-dev unzip
```

For TensorFlow:

```
# For Python 2.7
sudo apt-get install python-pip python-numpy swig python-dev
```

```
sudo pip install wheel

# For Python 3.3+
sudo apt-get install python3-pip python3-numpy swig python3-dev
sudo pip3 install wheel
```

To be able to take advantage of certain optimization flags:

```
sudo apt-get install gcc-4.8 g++-4.8
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 100
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8 100
```

Finally, for cleanliness, make a directory that will hold the Protobuf, Bazel, and TensorFlow repositories.

```
mkdir tf
cd tf
```

2. Install a Memory Drive as Swap for Compiling

In order to succesfully build TensorFlow, your Raspberry Pi needs a little bit more memory to fall back on. Fortunately, this process is pretty straightforward. Grab a USB storage drive that has at least 1GB of memory. I used a flash drive I could live without that carried no important data. That said, we're only going to be using the drive as swap while we compile, so this process shouldn't do too much damage to a relatively new USB drive.

First, put insert your USB drive, and find the `/dev/xxx` path for the device.

```
sudo blkid
```

As an example, my drive's path was `/dev/sda1`

Once you've found your device, unmount it by using the `umount` command.

```
sudo umount /dev/XXX
```

Then format your device to be swap:

```
sudo mkswap /dev/XXX
```

If the previous command outputted an alphanumeric UUID, copy that now. Otherwise, find the UUID by running `blkid` again. Copy the UUID associated with `/dev/xxx`

```
sudo blkid
```

Now edit your `/etc/fstab` file to register your swap file. (I'm a Vim guy, but Nano is installed by default)

```
sudo nano /etc/fstab
```

On a separate line, enter the following information. Replace the X's with the UUID (without quotes)

```
UUID=XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX none swap sw,pri=5 0 0
```

Save `/etc/fstab` , exit your text editor, and run the following command:

```
sudo swapon -a
```

If you get an error claiming it can't find your UUID, go back and edit `/etc/fstab` . Replace the `UUID=xxx. .` bit with the original `/dev/xxx` information.

```
sudo nano /etc/fstab
```

```
# Replace the UUID with /dev/XXX
/dev/XXX none swap sw,pri=5 0 0
```

Alright! You've got swap! Don't throw out the `/dev/xxx` information yet- you'll need it to remove the device safely later on.

3. Build Bazel

To build [Bazel](#), we're going to need to download a zip file containing a distribution archive. Let's do that now and extract it into a new directory called `bazel` :

```
wget https://github.com/bazelbuild/bazel/releases/download/0.4.5/bazel-0.4.5-dist.zip
unzip -d bazel bazel-0.4.5-dist.zip
```

Once it's done downloading and extracting, we can move into the directory to make a few changes:

```
cd bazel
```

Before building Bazel, we need to set the `javac` maximum heap size for this job, or else we'll get an `OutOfMemoryError`. To do this, we need to make a small addition to `bazel/scripts/bootstrap/compile.sh` . (Shout-out to [@SangManLINUX](#) for [pointing this out](#)..

```
nano scripts/bootstrap/compile.sh
```

Move down to line 117, where you'll see the following block of code:

```
run "${JAVAC}" -classpath "${classpath}" -sourcepath "${sourcepath}" \
  -d "${output}/classes" -source "$JAVA_VERSION" -target "$JAVA_VERSION" \
  -encoding UTF-8 "@${paramfile}"
```

At the end of this block, add in the `-J-Xmx500M` flag, which sets the maximum size of the Java heap to 500 MB:

```
run "${JAVAC}" -classpath "${classpath}" -sourcepath "${sourcepath}" \
  -d "${output}/classes" -source "$JAVA_VERSION" -target "$JAVA_VERSION" \
  -encoding UTF-8 "@${paramfile}" -J-Xmx500M
```

Finally, we have to add one thing to `tools/cpp/cc_configure.bzl` - open it up for editing:

```
nano tools/cpp/cc_configure.bzl
```

Place the line `return "arm"` around line 133 (at the beginning of the `_get_cpu_value` function):

```
...
"""Compute the cpu_value based on the OS name."""
return "arm"
...
```

In newer Bazel versions, the `_get_cpu_value` function will be found in the file `tools/cpp/lib_cc_configure.bzl` , and the same modification is required.

Now we can build Bazel! *Note: this also takes some time.*

```
sudo ./compile.sh
```

When the build finishes, you end up with a new binary, `output/bazel` . Copy that to your `/usr/local/bin` directory.

```
sudo cp output/bazel /usr/local/bin/bazel
```

To make sure it's working properly, run `bazel` on the command line and verify it prints help text. Note: this may take 15-30 seconds to run, so be patient!

```
bazel
```

```
Usage: bazel <command> <options> ...
```

```
Available commands:
analyze-profile    Analyzes build profile data.
build              Builds the specified targets.
canonicalize-flags Canonicalizes a list of bazel options.
clean              Removes output files and optionally stops the server.
dump              Dumps the internal state of the bazel server process.
fetch              Fetches external repositories that are prerequisites to the targets.
help              Prints help for commands, or the index.
info              Displays runtime info about the bazel server.
```

mobile-install	Installs targets to mobile devices.
query	Executes a dependency graph query.
run	Runs the specified target.
shutdown	Stops the bazel server.
test	Builds and runs the specified test targets.
version	Prints version information for bazel.

Getting more help:

```
bazel help <command>
    Prints help and options for <command>.
bazel help startup_options
    Options for the JVM hosting bazel.
bazel help target-syntax
    Explains the syntax for specifying targets.
bazel help info-keys
    Displays a list of keys used by the info command.
```

Move out of the bazel directory, and we'll move onto the next step.

```
cd ..
```

4. Compiling TensorFlow

First things first, clone the TensorFlow repository and move into the newly created directory.

```
git clone --recurse-submodules https://github.com/tensorflow/tensorflow.git
cd tensorflow
```

Note: if you're looking to build to a specific version or commit of TensorFlow (as opposed to the HEAD at master), you should git checkout it now.

Once in the directory, we have to write a nifty one-liner that is incredibly important. The next line goes through all files and changes references of 64-bit program implementations (which we don't have access to) to 32-bit implementations. Neat!

```
grep -Rl 'lib64' | xargs sed -i 's/lib64/lib/g'
```

Updating tensorflow/core/platform/platform.h and WORKSPACE as listed in the previous version is no longer needed with the latest version of tensorflow.

- the IS_MOBILE_PLATFORM check now includes a specific check for the Raspberry
- numeric/1.2.6 is no longer listed WORKSPACE

Finally, we have to replace the eigen version dependency. The version included in the current tensorflow version may result in an error (near the end of the build):

```
ERROR: /mnt/tensorflow/tensorflow/core/kernels/BUILD:2128:1: C++ compilation of rule '//tensorflow/core/kernels:svd_c
...com.google.devtools.build.lib.shell.BadExitStatusException: Process exited with status 1.
...
external/eigen_archive/Eigen/src/Jacobi/Jacobi.h:359:55: error: 'struct Eigen::internal::conj_helper<__vector(4) __bu
```

to resolve this

```
sudo nano tensorflow/workspace.bzl
```

Replace the following

```
native.new_http_archive(
    name = "eigen_archive",
    urls = [
        "http://mirror.bazel.build/bitbucket.org/eigen/eigen/get/f3a22f35b044.tar.gz",
        "https://bitbucket.org/eigen/eigen/get/f3a22f35b044.tar.gz",
    ],
    sha256 = "ca7beac153d4059c02c8fc59816c82d54ea47fe58365e8aded4082ded0b820c4",
    strip_prefix = "eigen-eigen-f3a22f35b044",
    build_file = str(Label("//third_party:eigen.BUILD")),
)
```

with

```
native.new_http_archive(  
    name = "eigen_archive",  
    urls = [  
        "http://mirror.bazel.build/bitbucket.org/eigen/eigen/get/d781c1de9834.tar.gz",  
        "https://bitbucket.org/eigen/eigen/get/d781c1de9834.tar.gz",  
    ],  
    sha256 = "a34b208da6ec18fa8da963369e166e4a368612c14d956dd2f9d7072904675d9b",  
    strip_prefix = "eigen-eigen-d781c1de9834",  
    build_file = str(Label("//third_party:eigen.BUILD")),  
)
```

Reference: <https://stackoverflow.com/questions/44418657/how-to-build-eigen-with-arm-neon-compile-error-for-tensorflow>

These options have changed with exception of jemalloc use No for all Now let's configure the build:

```
./configure
```

```
Please specify the location of python. [Default is /usr/bin/python]: /usr/bin/python  
Please specify optimization flags to use during compilation when bazel option "--config=opt" is specified [Default is  
Do you wish to use jemalloc as the malloc implementation? [Y/n] Y  
Do you wish to build TensorFlow with Google Cloud Platform support? [y/N] N  
Do you wish to build TensorFlow with Hadoop File System support? [y/N] N  
Do you wish to build TensorFlow with the XLA just-in-time compiler (experimental)? [y/N] N  
Please input the desired Python library path to use. Default is [/usr/local/lib/python2.7/dist-packages]  
Do you wish to build TensorFlow with OpenCL support? [y/N] N  
Do you wish to build TensorFlow with CUDA support? [y/N] N
```

Note: if you want to build for Python 3, specify /usr/bin/python3 for Python's location and /usr/Local/Lib/python3.5/dist-packages for the Python library path.

Bazel will now attempt to clean. This takes a really long time (and often ends up erroring out anyway), so you can send some keyboard interrupts (CTRL-C) to skip this and save some time.

Now we can use it to build TensorFlow! **Warning: This takes a really, really long time. Several hours.**

```
bazel build -c opt --copt="-mfpu=neon-vfpv4" --copt="-funsafe-math-optimizations" --copt="-ftree-vectorize" --copt="-
```

Note: I toyed around with telling Bazel to use all four cores in the Raspberry Pi, but that seemed to make compiling more prone to completely locking up. This process takes a long time regardless, so I'm sticking with the more reliable options here. If you want to be bold, try using --Local_resources 1024,2.0,1.0 or --Local_resources 1024,4.0,1.0

When you wake up the next morning and it's finished compiling, you're in the home stretch! Use the built binary file to create a Python wheel.

```
bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg
```

And then install it!

```
sudo pip install /tmp/tensorflow_pkg/tensorflow-1.1.0-cp27-none-linux_armv7l.whl
```

5. Cleaning Up

There's one last bit of house-cleaning we need to do before we're done: remove the USB drive that we've been using as swap.

First, turn off your drive as swap:

```
sudo swapoff /dev/XXX
```

Finally, remove the line you wrote in /etc/fstab referencing the device

```
sudo nano /etc/fstab
```

Then reboot your Raspberry Pi.

And you're done! You deserve a break.

References

1. [Building TensorFlow for Jetson TK1](#) (an update to this post is available [here](#))
2. [Turning a USB Drive into swap](#)
3. [Safely removing USB swap space](#)

[Back to top](#)