# 1 How to: Run a neural network in Linux kernel

## 1.1 Overview

At high level, we need to do the following steps:

- "Freeze" the neural network trained by Tensorflow to Protobuf format.
- Compile the frozen graph using Tensorflow Ahead-of-Time compilation into an object file and a C++ header file.
- Convert the C++ header file to a C header file with no library dependency.
- Make the object file runnable in kernel.

While this tutorial targets the neural network that is used in Indigo (an NN-based congestion control scheme), it should work with minor modifications for any NN trained by Tensorflow.

## 1.2 Freezing the graph

Normally when doing training and inference in Python, the models are persisted in files called "checkpoints". Typically, a checkpoint generated from Python API consists of the following files:

- A `model.meta` and a `model.index`, the metadata of the graph.
- One or more files named `model.data-[k]-of-[n]`. The trained weights.

However, C++ API only accepts a different format. We will call it Protobuf format. Unlike checkpoints, it cannot be used for training. It typically consists of two files:

- A `graph.pb` file. Tensorflow's Python API can generate this file from a neural network.
- A `graph.config.pbtxt` file, which specifies the inputs and outputs of this graph. Unfortunately you need to hand-write this file yourself.

There are a few additional complexities to convert a checkpoint-format graph into Protobuf format. Specifically:

- Protobuf format does not support fancy constructs such as Python tuples. Unfortunately, RNN (which is used by Indigo) by default takes LSTM input states as Python tuples. There is a legacy interface `state_is_tuple=False` which allows you to pass in LSTM state as a crunched tensor. You have to modify Indigo's `models.py` to use this interface. Fortunately this does not affect the shape of trained weights, so you can still load the weights trained using older interface with no issue.
- You have to explicitly name all the inputs and outputs nodes. In Python those nodes can be referenced by Python variables so you don't have to name it, but this is no longer the case in Protobuf format. Input nodes can simply be named by adding `name="foo"` when constructing `tf.placeholders`. Output nodes must be given names by using an identity neuron as wrapper, like `named_node = tf.identity(unnamed_node, name="foo")`.
- You have to manually write a `graph.config.pbtxt` file that specifies all inputs and outputs node names (done in previous step) as well as their shapes. The file format itself is relatively straight-forward though, so it is not hard to replicate once you have an example.

## 1.3 Compiling the graph

Unfortunately you will have to compile the Tensorflow source code from scratch to do that, since currently the interface of AOT compilation is exposed as "a custom build target of Tensorflow source code".

- First of all, clone the tensorflow repo and checkout `r1.14` branch.
- Install Bazel `0.26.1`. A different version will not work.

- Install `gcc 8.3.0`. Unfortunately Ubuntu's default `gcc-7` (version `7.3.0`) or `gcc-8` (version `8.0.0`) will not work (you will hit a crash bug when you build the source code).
- Copy the `graph.pb` and `graph.config.pbtxt` file into the source code root directory.
- Add the following lines at the end of `BUILD` file in source code root directory.

```
load("//tensorflow/compiler/aot:tfcompile.bzl", "tf_library")
tf_library(
    name = "graph",
    graph = "graph.pb",
    config = "graph.config.pbtxt",
    cpp_class = "Graph"
)
```

- You need to workaround a bug in Tensorflow's build system before you can compile the graph into an object file. The patch can be found in Section 1.6.
- Run `./configure.py` to configure the hardware features your machine supports.
- Compile the graph by running the following command:

```
bazel build --config=opt :graph
```

  This will take a long time (30+min on my 8 hyperthread 32GB RAM machine) since it involves building the whole source code of Tensorflow. Unfortunately, you can not just leave the build there waiting for it to finish. GCC randomly crashes from time to time. You need to repeat the command until it succeeds.
- Go to `bazel-genfiles`. You should see two files: `graph.h` and `libgraph.pic.a`, the C++ header and the object file respectively.

## 1.4 Getting rid of library dependency

`graph.h` is a C++ header file, and it includes a large portion of files in the Tensorflow source tree. The official guide expects you to write your project in C++ inside Tensorflow source tree and build it as a custom build target, but this is of course not something we want: we need a stand-alone library, we need absolutely no library dependency, and we need to use C, not C++.

Fortunately, it turns out that only a very small amount of code in the includes are actually needed. So I wrote a tool named `tf_converter` which automatically converts this C++ header file into a C header file with no outside dependency. I re-implemented the necessary logic to invoke the object file in C.

- Clone the `tf_converter` repo.
- Copy the `graph.h` and `libgraph.pic.a` into the `inputs` folder.
- Run `make` to build the repo.
- Run `./main [desired C prefix] > indigo_nn.generated.h` to generate the C header.

The generated C header consists of the following functions:

- `[prefix]_constructor`: the constructor.
- `[prefix]_destructor`: the destructor.
- `[prefix]_get_address_for_argument(int index)`: returns the pointer to populate data for the given argument ordinal.
- `[prefix]_run`: execute the NN after populating all arguments.
- `[prefix]_get_address_for_result(int index)`: returns the pointer that points to the output of the NN for the given output ordinal.

## 1.5 Invoke the NN in Linux kernel

Finally, we are almost ready to invoke the neural network in Linux kernel. But again, there are a few additional complexities involved:

- Using floating-points and vectorized instructions is generally disallowed in kernel, because the kernel restores the FPU registers in a lazy manner. To use them, you must wrap your code with `kernel_fpu_begin()` and `kernel_fpu_end()`, like this:

  ```
  kernel_fpu_begin();
  .. use floating points or vectorized instructions ..
  kernel_fpu_end();
  ```

  You must do this to invoke the NN of course.

- In user-space code, the stack pointer is supposed to be always 16-byte aligned at the start of a function call. GCC compiler also makes use of this fact to do optimizations, for example, by using aligned AVX instruction (instead of their unaligned variant) for better performance. However, in kernel this is not true, as some assembly codes does not respect this contract. This results in Tensorflow's compiled object file to crash in the kernel due to using aligned AVX instruction on unaligned address. To fix this, we must do manually write inline assembly code to invoke the object file. We first align the stack if the stack is misaligned, then call the function using the expected calling convention (the System V convention), and finally after the call restore the stack pointer to its original position (See Section 1.6).

- Normally a kernel module is not expected to use floating points (some architectures do not even support that!). To use floating points, you need to add the following line in `Makefile`:

  ```
  ccflags-y := -mhard-float -msse
  ```

- The object file generated by Tensorflow may use symbols in the math library, which is not available in kernel. To fix this, we need to supply implementations for all such symbols. In Indigo's neural network, the generated object file used `floorf` and `expf` (you can figure this out by either running `nm libgraph.pic.a`, or checking the link errors when you build the kernel module). We need to sneak into the `glibc` source code to steal out the relavent code that implements the functionality (See Section 1.6).

## 1.6 Putting Everything Together: A Step-By-Step Guide

I have made a repo which includes everything needed to convert the NN in indigo into a simple kernel module which invokes it in kernel. Below is a step-by-step guide.

- First of all, clone the repo.

  ```
  git clone https://github.com/sillycross/indigo-in-kernel
  cd indigo-in-kernel
  ```

- Let's do the first step (Section 1.2), which converts the graph to Protobuf format. Clone the original Indigo repo, and checkout our target branch.

  ```
  git clone https://github.com/StanfordSNR/indigo
  cd indigo
  git checkout 463d89b09699a57bfdfbae351646df6a60040b90
  git checkout -b indigo_kernel
  ```

  Now apply the patch `indigo-patch.txt` which contains necessary changes to `models.py` as well as code that converts the NN to Protobuf format.

  ```
  git apply ../indigo-patch.txt
  ```

  Now generate the `graph.pb` file:

```
python3 dagger/gen.py
```

You should now see a file named `graph.pb` in the root folder of indigo repo.

- Now we are ready to compile the graph (Section 1.3). First of all, make sure you have the correct version of `Bazel` and `gcc` installed:

```
bazel version      # should get "Build label: 0.26.1"
gcc -v             # should get "gcc version 8.3.0"
```

Now, clone the Tensorflow source code repo, and checkout the target branch.

```
cd ..  # go back to indigo-in-kernel root folder
git clone https://github.com/tensorflow/tensorflow
cd tensorflow
git checkout r1.14
```

Unfortunately there is a bug (build failure with `no such package "tools/target_cpu"`), which you have to work around. Apply the patch which contains the workaround as well as necessary changes to the `BUILD` file.

```
git apply ../tensorflow-patch.txt
```

Copy the `graph.pb` file we just generated, and the `graph.config.pbtxt` file (which is prepared by me) into the Tensorflow source tree:

```
cp ../graph.config.pbtxt .
cp ../indigo/graph.pb .
```

Configure Tensorflow:

```
./configure
```

Compile the graph using Tensorflow AOT compilation feature, retry if it crashes:

```
bazel build --config=opt :graph
```

You should see two files: `graph.h` and `libgraph.pic.a` in `bazel-genfiles`. Copy them to the root of our repo so we can access them easier. We are now ready to do the next step.

```
cp bazel-genfiles/graph.h ..
cp bazel-genfiles/libgraph.pic.a ..
chmod 644 ../libgraph.pic.a
```

- We are now ready to do the steps in Section 1.4 that converts the C++ header file to a C header file with no outside dependency. First, go to the `tf_converter` directory.

```
cd ../tf_converter
```

Copy `graph.h` and `libgraph.pic.a` into the `input` folder.

```
mkdir input
cp ../graph.h input
cp ../libgraph.pic.a input
```

Build the converter tool, and generate the stand-alone C header file:

```
make
./main indigo_nn > indigo_nn.generated.h
```

- Now, using the generated stand-alone C header file and the object file, we can build our kernel module (Section 1.5). This kernel module simply runs the NN for 10 times on the same input vector $[0.1, 0.2, \cdots, 0.9]$ and outputs the results.

  ```
  cd ../kmodule
  cp ../tf_converter/indigo_nn.generated.h .
  cp ../libgraph.pic.a .
  make
  ```

  Check `indigo_utils.c` for the code that properly aligns the stack before invoking the generated function. `math.c` contains implementations of `floorf` and `expf` stolen from `glibc` source code.

  Finally, we can run the module in kernel. Open a new terminal window and run

  ```
  dmesg -w
  ```

  to monitor the real-time dmesg output. Now, load and then unload the kernel module by

  ```
  sudo insmod indigo.ko
  sudo rmmod indigo.ko
  ```

  You should see the following output in dmesg:

  ```
  [228193.438910] Running neural network in kernel..
  [228193.438960] Iteration 0: output is 0.16509 0.26258 0.07892 0.46645 0.02693
  [228193.438978] Iteration 1: output is 0.32704 0.33405 0.03967 0.28155 0.01766
  [228193.438995] Iteration 2: output is 0.31607 0.45213 0.02839 0.18348 0.01991
  [228193.439013] Iteration 3: output is 0.26527 0.53841 0.02520 0.14351 0.02759
  [228193.439029] Iteration 4: output is 0.23542 0.56457 0.02471 0.13906 0.03621
  [228193.439045] Iteration 5: output is 0.21033 0.56497 0.02445 0.15352 0.04670
  [228193.439062] Iteration 6: output is 0.18218 0.55026 0.02379 0.18257 0.06118
  [228193.439079] Iteration 7: output is 0.15099 0.52142 0.02277 0.22422 0.08058
  [228193.439095] Iteration 8: output is 0.12036 0.48114 0.02139 0.27350 0.10359
  [228193.439111] Iteration 9: output is 0.09390 0.43560 0.01973 0.32337 0.12737
  [228197.562219] Unregistering Kernel Module..
  ```

  The output changes with each iteration, despite that the input vector is the same. This is expected, because LSTM has an internal state which changes internally after each inference. If you run the Python version Indigo with the same input vectors, you should see the same results.

## 1.7 Additional Notes

Below are some misc notes on the challenges and pitfalls I encountered when developing the kernel module.

- All functions in the congestion control algorithm are executed with preemption off, so it must not call functions that may give out CPU, otherwise you may have random kernel freeze. This means you can only use `kmalloc` with `GFP_NOWAIT` (not `GFP_KERNEL`). Using `vmalloc` always results in a freeze though I'm not sure why.
- Using floating points require the `-msse` flag, but that would also result in your compiler optimization generate SSE instructions at disallowed places (outside `kernel_fpu_begin/end` section). My workaround for this is to separate functions that use and do not use floating points into different files, and only add `-msse` flag for the files that use floating points.