Haskell Tensor Flow Guide

Table of Contents

Section 1: Installing Haskell Tensor Flow

In the first part of this guide, we'll go over how to use Stack in conjunction with some other tools in order to get the basic Tensor flow libraries up and running. You can check out the README on the Github repository for additional assistance. I've performed the instructions on both Mac and Linux. If you experience any issues with this (or any other) part of the guide, email me at james@mondaymorninghaskell.me so I can help you and make this guide better!

First off, Tensor Flow is a Google product, and hence it depends on a number of other Google dependencies. These dependencies are `protobuf`, `snappy`, and `tensorflow` itself. If you're on a Mac, one easy way to get these would be to clone the Haskell Tensor Flow repository and use the `tools/install_osx_dependencies` script. If you're on Linux, this isn't an option, so you'll have to install these three things manually. The instructions I include are mostly from my test run on Linux, but they mostly follow what happens in the Mac OS instructions as well.

Installing Protobuf

The `protobuf` library implements Google's data interchange format. It allows different programs in totally different languages to communicate with each other in a sane way. In particular, we need this so that our Haskell code can communicate with the low level C code that Tensor Flow uses to perform all the math operations.

First, you have to retrieve the `protobuf` source (I'm using version 3.2.0, but be sure to check for later versions):

```
>> curl -OL https://github.com/google/protobuf/releases/download/
v3.2.0/protoc-3.2.0-linux-x86_64.zip
```

Next, unzip it:

```
>> unzip protoc-3.2.0-linux-x86_64.zip -d protoc3
```

Now you need to move the binary executables and included headers to your path where they can be found:

```
>> sudo mv protoc3/bin/* /usr/local/bin/
>> sudo mv protoc3/include/* /usr/local/include/
```

You'll know you've succeeded if the `protoc` command (without any arguments) outputs `Missing input file,` rather than something like `command not found.`

Installing Snappy

Snappy is a compression library. You can install it with homebrew on Mac (`brew install snappy`). However, it's also pretty painless to download and compile from source. Follow the make instructions in the README. Then you'll just need to move the relevant files onto your path. Here's the full instruction set in Linux (if you're installing from source on Mac, it'll look very similar).

```
>> sudo apt install cmake
>> git clone https://github.com/google/snappy.git
>> git checkout tags/1.1.6
>> mkdir build && cd build
>> cmake ../
>> make
>> sudo cp ../snappy.h /usr/local/include/
>> sudo cp snappy-stubs-public.h /usr/local/include/
>> sudo cp ../snappy-sinksource.h /usr/local/include/
>> sudo cp libsnappy.so /usr/local/lib/
```

Install Tensor Flow

Naturally, you also have to install Tensor Flow itself. By this, I mean the low-level machinery that actually performs all the math and builds up your graph. Whether you program in Python or in Haskell (or any other language), you're really just writing a Layer on top of this. You'll want to download the latest version for your system from Google APIs. Then unzip it, and you'll be able to copy the dynamic library onto your path so it can be seen. Here are the commands from the script for installing the Mac OS dependencies:

```
>> curl https://storage.googleapis.com/tensorflow/libtensorflow/
libtensorflow-cpu-darwin-x86_64-1.0.0.tar.gz > libtensorflow.tar.gz
>> sudo tar zxf libtensorflow.tar.gz -C /usr/local
>> rm libtensorflow.tar.gz
>> sudo mv /usr/local/lib/libtensorflow.so /usr/local/lib/
libtensorflow.dylib
>> sudo install_name_tool -id libtensorflow.dylib /usr/local/lib/
libtensorflow.dylib
```

Here's the process from scratch on Linux. Note the `ldconfig` command, which links the dynamic library. You do this instead of actually changing it to a `.dylib` file (that concept only exists on Mac OS).

```
>> curl https://storage.googleapis.com/tensorflow/libtensorflow/
libtensorflow-cpu-linux-x86_64-1.0.0.tar.gz > libtensorflow.tar.gz
>> sudo tar zxf libtensorflow.tar.gz -C /usr/local
>> sudo ldconfig
```

Bringing Tensor Flow into Haskell with Stack

Now that you've got all the dependencies installed, you need to start a Haskell project with Stack and bring the Haskell Tensor Flow library in as a dependency. The library is still in its early stages, so this is not quite so straightforward. There are a couple early versions on Hackage, but they don't include a lot of important recent work. So my recommendation is to use the bleeding edge version from Github. This guide and the blog posts that accompany it were written from commit `4ab9cb9cf274e88eababfaba85103e3890a96afc`.

To get it, we'll use the feature of Stack that allows us to use a particular Github link and commit as a package within our project. The repository uses a multi-package structure. This means you'll have to specify each individual subdirectory in the `subdirs` section of the definition so that so you can access the code. All in all, you'll want this section as a "package" in your `stack.yaml` file:

```
- location:
    git: https://github.com/tensorflow/haskell.git
    commit: 4ab9cb9cf274e88eababfaba85103e3890a96afc
  subdirs:
  - tensorflow
  - tensorflow-ops
  - tensorflow-core-ops
  - tensorflow-logging
  - tensorflow-proto
  - tensorflow-opgen
  - tensorflow-test
  - tensorflow-mnist
  - tensorflow-mnist-input-data
  extra-dep: true
```

In addition to that, you also need some extra dependencies. These are essentially Haskell bindings to some of the programs we installed earlier that don't necessarily live within the "resolvers" that Stackage uses. Some of these are only necessary if you want to do logging, but it's extremely frustrating to have to stop and figure that out in the middle of development, so I highly recommend you start with them all.

```
extra-deps:
- proto-lens-protobuf-types-0.2.1.0
- snappy-0.2.0.2
- snappy-framing-0.1.1
- tensorflow-logging-0.1.0.0
- tensorflow-records-0.1.0.0
- tensorflow-records-conduit-0.1.0.0
```

Finally, you have to make sure you update the `extra-include-dir` and `extra-lib-dir` fields to contain the directories where you put the header and library files for Snappy and Tensorflow.

```
extra-include-dirs:
- /usr/local/include
```

```
extra-lib-dirs:
- /usr/local/lib
```

For the last step, all you need to do is include the relevant dependencies in your `.cabal` file. Again, I've included a couple things here, such as `mnist` and `logging`, that aren't necessary for the core. But some of the blog material will require them. Here's what the build-depends section looks like:

```
  build-depends:       base >= 4.7 && < 5
                     , tensorflow
                     , tensorflow-ops
                     , tensorflow-core-ops
                     , tensorflow-logging
                     , tensorflow-mnist
                     , proto-lens
```

And that should be it! You should now be able to make a file and import the tensor flow files without any compiler errors!

```
module Lib where

import TensorFlow.Core
import TensorFlow.Logging
import TensorFlow.GenOps.Core
…
```

Congratulations, you can now run Tensor Flow in Haskell!

Section 2: Overview of Files

In this section, we'll got over the most important files in the Tensor Flow library and what functions and types they export.

TensorFlow.Core

This module contains a lot of the most important pieces of Tensor Flow functionality. It's a good place to look to start. It has, for instance, many important types like the `Tensor` type and the `Build` monad. It also has functions for actually running tensors: `run` and `runWithFeeds`.

TensorFlow.Ops

TensorFlow.Ops contains all the basic mathematical operations you'll want to perform. For example, you'll find things like tensor addition, multiplication, and matrix multiplication. You'll also see useful conversion functions like `scalar, vector,` and `cast`.

TensorFlow.Variable

The `Variable` module naturally contains operations related to Tensor Flow variables. Examples include the `Variable` type, `initializedVariable`, and the `readValue` function that allows operations to be performed on variables.

TensorFlow.Session

The `Session` module has the `Session` type as well as the `runSession` function, which you'll need to actually run most Tensor Flow programs.

TensorFlow.Minimize

This module contains optimizers that you'll use to train your algorithms. It has, for instance, the basic `gradientDescent` optimizer as well as the more sophisticated `adam` optimizer. You'll generally use the `minimizeWith` function from this module, combined with one of the optimizers.

TensorFlow.Logging

This module contains functionality related to logging events for Tensor Board. You'll find functions like `logGraph, logSummary, and withEventWriter.`

TensorFlow.GenOps.Core

So this module is a little tricky. It does not exist concretely, in that the file does not live within the repository. Hence it doesn't actually have any documentation. The file is generated for you at compile time by the `tensorflow-opgen` package. It takes all the low-level C functions that do not have Tensor Flow specific implementations and turns them into Haskell functions you can use.

Some of the C functions have optional arguments with default values. This makes them rather awkward to use from a Haskell perspective, and not especially type safe. Here's an example of using optional parameters to call the 2D convolution function:

```
let conv = conv2D' convAttrs …
where
  convStridesAttr = opAttr "strides" .~ ([1,1,1,1] :: [Int64])
  paddingAttr = opAttr "padding" .~ ("SAME" :: ByteString)
  dataFormatAttr = opAttr "data_format" .~ ("NHWC" :: ByteString)
  convAttrs = convStridesAttr . paddingAttr . dataFormatAttr
```

These files should be enough to get you started! After a little while, you'll get used to which files you need to import for which tasks. The example at the end uses explicit imports, so you can learn from that as well.

Section 3: Important Types in Haskell Tensor Flow

In this section, we'll focus on two main categories of types within Haskell Tensor Flow. First, we'll look at the Tensor type, and then we'll examine the two most important monads, `Build` and `Session`. If this section seems confusing, I encourage you to move onto Section 4, try out some of the code examples, and THEN come back to this section after you've seen these types in action.

Tensor Types

So many of the objects you will create in your Tensor Flow Graph are of type `Tensor v a`. What does this mean exactly? The `Tensor` type is parameterized by two other types. The first type parameter focuses on the "state" of the tensor. The second type focuses on the data that is stored within the Tensor. We'll start by focusing on the data parameter.

Tensors mostly store numerical values. Thus the most common types you will see for this second type parameter are things like `Float` or `Int64`. There are a couple other types that can show up from time to time. For instance, we can make a "summary tensor" that uses a `ByteString` for its data. This byte string represents an event we can log. It is also possible to store boolean values in tensors.

There is a decent degree of type safety with these. Most of the mathematical operations between two tensors (add, subtract, etc.) won't typecheck when called on tensors containing bytestrings or boolean values. The following results in a type error:

```
let n1 = constant (Shape [1]) [True]
let n2 = constant (Shape [1]) [4 :: Float]
let additionNode = n1 `add` n2


— Error:
Couldn't match type ‘TensorFlow.Types.TypeError Bool’
                    with ‘TensorFlow.Types.ExcludedCase’
arising from a use of ‘add’
```

Note that mathematical operations typically require the same underlying value type in the tensor. For instance, the following will also not typecheck:

```
let n1 = constant (Shape [1]) [1 :: Int32]
let n2 = constant (Shape [1]) [4 :: Float]
let additionNode = n1 `add` n2
```

You'll need to fix this by changing the first line to use a `Float` instead. It can be a little tricky to try to change datatypes mid-stream. You'll likely have to get the values out and then re-encode them as a new tensor.

Tensor States

Now, Tensors can also have different states of existence, and this is what the first type parameter v indicates. There are three different states: `Build, Value,` and `Ref`. When you construct constant tensors, they'll be in the `Build` state. This refers to a tensor that cannot change its value, but has not been rendered in the TensorFlow graph yet. Note all the following examples will use the `ScopedTypeVariables` extension.

```
let (n1 :: Tensor Build Float) = constant (Shape [1]) [1 :: Float]
```

Tensors in the `Value` state already exist within the graph, so they are to some extent "fixed". When you are in the `Build` monad (see below), you can take any `Build` tensor and turn it into a `Value` tensor with the `render` function. You can also use the `renderValue` function if you're not sure state your tensor is currently in.

```
let (n1 :: Tensor Build Float) = constant (Shape [1]) [1 :: Float]
(rendered :: Tensor Value Float) <- render n1
```

When you create placeholders, they are in the `Value` state, so you need to make them from within a `Build` monad. Try not to get to confused at the distinction between the `Build` monad and the `Build` tensor state!

```
(n3 :: Tensor Value Float) <- placeholder [1]
```

You can take any tensor, rendered or not, and get the core expression out of it with the `expr` function. This will give you a `Build` tensor.

```
let (n1 :: Tensor Build Float) = constant (Shape [1]) [1 :: Float]
(rendered :: Tensor Value Float) <- render n1
(n3 :: Tensor Value Float) <- placeholder (Shape [1])
let (n4 :: Tensor Build Float) = expr rendered
let (n5 :: Tensor Build Float) = expr n3
```

The `Ref` state refers to tensors that have a particular state attaches to them. This typically refers to variables. You can easily convert them to `Tensor Value` items with the `value` function (no monad is needed).

```
value :: Tensor Ref a -> Tensor Value a
```

Unhelpfully, there are two initializedVariable functions in different modules, and their types are different. Using `TensorFlow.Ops.initializedVariable` will give you a `Tensor Ref a`. You can use this in operations as you would any tensor. Just note that the result will be a `Tensor Build`.

```
(opsVariable :: Tensor Ref Float) <-
  TensorFlow.Ops.initializedVariable 3
let (additionWithRef :: Tensor Build Float) = opsVariable `add` n1
-- Can also turn this into a "value" tensor.
let (opsValue :: Tensor Value Float) = value opsVariable
```

On the other hand, we can also use `TensorFlow.Variable.initializedVariable`. This gives something of type `Variable`, which wraps the tensor. You then need to use the `readValue` function in order to use it in computations. My examples will use this latter approach (mainly since it's the first approach I found in the docs), but I think the former approach is probably better.

```
(var :: Variable Float) <- TensorFlow.Variable.initializedVariable 3
let (readVar :: Tensor Build Float) = readValue var
-- You'll get a compile error if you try to add "var" directly.
let (additionWithVar :: Tensor Build Float) = readVar `add` n1
```

Build and Session Monads

There are two main monads you'll be using. The `Build` monad is what you'll use for actually constructing your Tensor Flow graph. You primarily need the `Build` monad to store the state of certain graph elements like placeholders and variables. Critically, you need to remember that the `Build` monad encapsulates the creation of tensors of ALL states, not just those in the `Build` state.

The `Session` monad stores the state of the Tensor Flow session. You'll need this to `run` your graph and actually get output. The session is also responsible for logging events. Note that `Session` belongs to the `MonadBuild` class, so we can call any `Build` function within the `Session` monad. The simple examples all do this. When you're building a more complex model though, it's generally a good idea to isolate the graph building logic from the session logic. Then you can just call the `build` function to lift your `Build` function into the `Session` monad.

```
mySimpleGraph :: Build (Tensor Build Float)
mySimpleGraph = do
  let myConstant = constant (Shape [1]) [1 :: Float]
  myVariable1 <- TensorFlow.Ops.initializedVariable 3
  myVariable2 <- TensorFlow.Ops.initializedVariable 5
  return $ myConstant `add` myVariable1 `add` myVariable2

mySession :: Session (Vector Float)
mySession = do
  graph <- build mySimpleGraph
  run graph
```

Section 4: Basic Examples

Here's a very basic example of a Tensor Flow application, with line-by-line comments explaining what's happening. Hopefully this is enough to get you off the ground and familiar with the basic concepts!

```
module Main where

import Control.Monad (replicateM_)
import qualified Data.Vector as Vector
import Data.Vector (Vector)

import TensorFlow.Core
  (Tensor, Value, feed, encodeTensorData, Scalar(..))
import TensorFlow.Ops
  (add, placeholder, sub, reduceSum, mul)
import TensorFlow.GenOps.Core (square)
import TensorFlow.Variable (readValue, initializedVariable, Variable)
import TensorFlow.Session (runSession, run, runWithFeeds)
import TensorFlow.Minimize (gradientDescent, minimizeWith)


-- Usage of the "basicExample" function. Should result in a tuple like:
-- (5, -1)
main :: IO ()
main = do
  results <- basicExample
    (Vector [1.0, 2.0, 3.0, 4.0])
    (Vector [4.0, 9.0, 14.0, 19.0])
  print results

-- We'll take two vectors (of equal size representing the inputs and
-- expected outputs of a linear equation.
basicExample :: Vector Float -> Vector Float -> IO (Float, Float)
basicExample xInput yInput = runSession $ do
  -- Everything below this ^^ takes place in the "Session" monad, which
  -- we run with "runSession".

  -- Get the sizes of out input and expected output
  let xSize = fromIntegral $ Vector.length xInput
  let ySize = fromIntegral $ Vector.length yInput

  -- Make a "weight" variable (slope of our line) with initial value 3
  (w :: Variable Float) <- initializedVariable 3
  -- Make a "bias" variable (y-intercept of line) with initial value 1
  (b :: Variable Float) <- initializedVariable 1

  -- Create "placeholders" with the size of our input and output
  (x :: Tensor Value Float) <- placeholder [xSize]
  (y :: Tensor Value Float) <- placeholder [ySize]
```

```
-- Make our "model", which multiplies the weights by the input and
-- then adds the bias. Notice we use "readValue" to use operations on
-- variables. This is our "actual output" value.
let linear_model = ((readValue w) `mul` x) `add` (readValue b)

-- Find the difference between actual output and expected output y,
-- and then square it.
let square_deltas = square (linear_model `sub` y)

-- Get our "loss" function by taking the reduced sum of the above,
-- then "train" our model by using the gradient descent optimizer.
-- Notice we pass our weights and bias as the parameters that change.
let loss = reduceSum square_deltas
trainStep <- minimizeWith (gradientDescent 0.01) loss [w,b]

-- "Train" our model, but passing our input and output values as
-- "feeds" to fill in the placeholder values.
let trainWithFeeds = \xF yF -> runWithFeeds
      [ feed x xF
      , feed y yF
      ]
      trainStep

-- Run this training step 1000 times. Encode our input as
-- "TensorData"
replicateM_ 1000 $
  trainWithFeeds
    (encodeTensorData [xSize] xInput)
    (encodeTensorData [ySize] yInput))

-- "Run" our variables to see their learned values and return them
(Scalar w_learned, Scalar b_learned) <-
  run (readValue w, readValue b)
return (w_learned, b_learned)
```