Если вы видите что-то необычное, просто сообщите мне.

Haskell and Tensor Flow

AI systems are beginning to impact our lives more and more. It's a very important element to how software is being developed and will continue to be developed. But where does Haskell fit in this picture?

In this series, we'll go over the basic concepts of Tensor Flow, one of the most easier machine learning frameworks to pick. We'll first try it out in Python (the most common language for TF), and then we'll translate our code to Haskell. For some help on actually installing the Haskell Tensor Flow library so you can write your own code, make sure to download our Haskell Tensor Flow Guide!

If you're already a bit familiar with these bindings, you can move on to part 2. We'll see how we can apply some advanced type system tricks to actually make our Tensor Flow code safer!

*Note this series will not be a general introduction to the concept of machine learning. There is a fantastic series on Medium about that called Machine Learning is Fun! If you're interested in learning the basic concepts, I highly recommend you check out part 1 of that series. Many of the ideas in my own article series will be a lot clearer with that background.

TENSORS Tensor Flow is a great name because it breaks the library down into the two essential concepts. First up are tensors. These are the primary vehicle of data representation in Tensor Flow. Low-dimensional tensors are actually quite intuitive. But there comes a point when you can't really visualize what's going on, so you have to let the theoretical idea guide you.

In the world of big data, we represent everything numerically. And when you have a group of numbers, a programmer's natural instinct is to put those in an array.

```
[1.0, 2.0, 3.0, 6.7]
```

Now what do you do if you have a lot of different arrays of the same size and you want to associate them together? You make a 2-dimensional array (an array of arrays), which we also refer to as a

matrix.

```
[[1.0, 2.0, 3.0, 6.7],
[5.0, 10.0, 3.0, 12.9],
[6.0, 12.0, 15.0, 13.6],
[7.0, 22.0, 8.0, 5.3]]
```

Most programmers are pretty familiar with these concepts. Tensors take this idea and keep extending it. What happens when you have a lot of matrices of the same size? You can group them together as an array of matrices. We could call this a three-dimensional matrix. But "tensor" is the term we'll use for this data representation in all dimensions.

Every tensor has a degree. We could start with a single number. This is a tensor of degree 0. Then a normal array is a tensor of degree 1. Then a matrix is a tensor of degree 2. Our last example would be a tensor of degree 3. And you can keep adding these on to each other, ad infinitum.

Every tensor has a shape. The shape is an array representing the dimensions of the tensor. The length of this array will be the degree of the tensor. So a number will have the empty list as its shape. An array will have a list of length 1 containing the length of the array. A matrix will have a list of length 2 containing its number of rows and columns. And so on. There are a few different ways we can represent tensors in code, but we'll get to that in a bit.

GO WITH THE FLOW

The second important concept to understand is how Tensor Flow performs computations. Machine learning generally involves simple math operations. A lot of simple math operations. Since the scale is so large, we need to perform these operations as fast as possible. And we need to use software and hardware that is optimized for these specific tasks. This necessitates having a low-level code representation of what's going on. This is easier to achieve in a language like C, instead of Haskell or Python.

We could have the bulk of our code in Haskell, but perform the math in C using a Foreign Function Interface. But these interfaces have a large overhead, so this is likely to negate most of the gains we get from using C.

Tensor Flow’s solution to this problem is that we first build up a graph describing all our computations. Then once we have described that, we "run" our graph using a "session". Thus it performs the entire language conversion process at once, so the overhead is lower.

If this sounds familiar, it's because this is how actions tend to work in Haskell (in some sense). We can, for instance, describe an IO action. And this action isn’t a series of commands that we execute the moment they show up in the code. Rather, the action is a description of the operations that our program will perform at some point. It’s also similar to the concept of Effectful programming.

So what does our computation graph look like? Well, each tensor is a node. Then we can make other nodes for "operations" that take tensors as input. For instance, we can "add" two tensors together, and this is another node. We’ll see in our example how we build up the computational graph, and then run it.

#CODING TENSORS So at this point we should start examining how we actually create tensors in our code. We’ll start by looking at how we do this in Python, since the concepts are a little easier to understand that way. There are three types of tensors we’ll consider. The first are "constants". These represent a set of values that do not change. We can use these values throughout our model training process, and they'll be the same each time. Since we define the values for the tensor up front, there’s no need to give any size arguments. But we will specify the datatype that we’ll use for them.

```
import tensorflow as tf

node1 = tf.constant(3.0, dtype=tf.float32)
node2 = tf.constant(4.0, dtype=tf.float32)
```

Now what can we actually do with these tensors? Well for a quick sample, let’s try adding them. This creates a new node in our graph that represents the addition of these two tensors. Then we can "run" that addition node to see the result. To encapsulate all our information, we’ll create a "Session":

```
import tensorflow as tf

node1 = tf.constant(3.0, dtype=tf.float32)
node2 = tf.constant(4.0, dtype=tf.float32)
additionNode = tf.add(node1, node2)
```

```
sess = tf.Session()
result = sess.run(additionNode)
print result

"""
Output:
7.0
"""
```

The next type of tensors are placeholders. These are values that we change each run. Generally, we will use these for the inputs to our model. By using placeholders, we'll be able to change the input and train on different values each time. When we "run" a session, we need to assign values to each of these nodes.

We don’t know the values that will go into a placeholder, but we still assign the type of data at construction. We can also assign a size if we like. So here’s a quick snippet showing how we initialize placeholders. Then we can assign different values with each run of the application. Even though our placeholder tensors don’t have values, we can still add them just as we could with constant tensors.

```
node1 = tf.placeholder(tf.float32)
node2 = tf.placeholder(tf.float32)
adderNode = tf.add(node1, node2)

sess = tf.Session()
result1 = sess.run(adderNode, {node1: 3, node2: 4.5 })
result2 = sess.run(adderNode, {node1: 2.7, node2: 8.9 })
print(result1)
print(result2)

"""
Output:
7.5
11.6
"""
```

The last type of tensor we’ll use are variables. These are the values that will constitute our "model". Our goal is to find values for these parameters that will make our model fit the data well.

We'll supply a data type, as always. In this situation, we'll also provide an initial constant value. Normally, we'd want to use a random distribution of some kind. The tensor won't actually take on its value until we run a global variable initializer function. We'll have to create this initializer and then have our session object run it before we get going.

```
w = tf.Variable([3], dtype=tf.float32)
b = tf.Variable([1], dtype=tf.float32)

sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
```

Now let's use our variables to create a "model" of sorts. For this article we'll make a simple linear model. Let's create additional nodes for our input tensor and the model itself. We'll let w be the weights, and b be the "bias". This means we'll construct our final value by w*x + b, where x is the input.

```
w = tf.Variable([3], dtype=tf.float32)
b = tf.Variable([1], dtype=tf.float32)
x = tf.placeholder(dtype=tf.float32)
linear_model = w * x + b
```

Now, we want to know how good our model is. So let's compare it to y, an input of our expected values. We'll take the difference, square it, and then use the reduce_sum library function to get our "loss". The loss measures the difference between what we want our model to represent and what it actually represents.

```
w = tf.Variable([3], dtype=tf.float32)
b = tf.Variable([1], dtype=tf.float32)
x = tf.placeholder(dtype=tf.float32)
linear_model = w * x + b
y = tf.placeholder(dtype=tf.float32)
squared_deltas = tf.square(linear_model - y)
loss = tf.reduce_sum(squared_deltas)
```

Each line here is a different tensor, or a new node in our graph. We'll finish up our model by using the built in GradientDescentOptimizer with a learning rate of 0.01. We'll set our training step as attempting to minimize the loss function.

```
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
```

Now we'll run the session, initialize the variables, and run our training step 1000 times. We'll pass a series of inputs with their expected outputs. Let's try to learn the line y = 5x - 1. Our expected output y values will assume this.

```
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
for i in range(1000):
    sess.run(train, {x: [1, 2, 3, 4], y: [4,9,14,19]})

print(sess.run([W,b]))
```

At the end we print the weights and bias, and we see our results!

```
[array([ 4.99999475], dtype=float32), array([-0.99998516], dtype=float32)]
```

So we can see that our learned values are very close to the correct values of 5 and -1!

REPRESENTING TENSORS IN HASKELL

So now at long last, I'm going to get into some of the details of how we apply these tensor concepts in Haskell. Like strings and numbers, we can't have this one "Tensor" type in Haskell, since that type could really represent some very different concepts. For a deeper look at the tensor types we're dealing with, check out our in depth guide.

In the meantime, let's go through some simple code snippets replicating our efforts in Python. Here's how we make a few constants and add them together. Do note the "overloaded lists" extension. It allows us to represent different types with the same syntax as lists. We use this with both Shape items and Vectors:

```
{-# LANGUAGE OverloadedLists #-}

import Data.Vector (Vector)
import TensorFlow.Ops (constant, add)
import TensorFlow.Session (runSession, run)

runSimple :: IO (Vector Float)
runSimple = runSession $ do
  let node1 = constant [1] [3 :: Float]
  let node2 = constant [1] [4 :: Float]
  let additionNode = node1 `add` node2
  run additionNode

main :: IO ()
main = do
  result <- runSimple
  print result

{-
Output:
[7.0]
-}
```

We use the constant function, which takes a Shape and then the value we want. We'll create our addition node and then run it to get the output, which is a vector with a single float. We wrap everything in the runSession function. This encapsulates the initialization and running actions we saw in Python.

Now suppose we want placeholders. This is a little more complicated in Haskell. We'll be using two placeholders, as we did in Python. We'll initialized them with the placeholder function and a shape. We'll take arguments to our function for the input values. To actually pass the parameters to fill in the placeholders, we have to use what we call a "feed".

We know that our adderNode depends on two values. So we'll write our run-step as a function that takes in two "feed" values, one for each placeholder. Then we'll assign those feeds to the proper nodes using the feed function. We'll put these in a list, and pass that list as an argument to runWithFeeds. Then, we wrap up by calling our run-step on our input data. We'll have to encode the raw vectors as tensors though.

```
import TensorFlow.Core (Tensor, Value, feed, encodeTensorData)
import TensorFlow.Ops (constant, add, placeholder)
import TensorFlow.Session (runSession, run, runWithFeeds)

import Data.Vector (Vector)

runPlaceholder :: Vector Float -> Vector Float -> IO (Vector Float)
runPlaceholder input1 input2 = runSession $ do
  (node1 :: Tensor Value Float) <- placeholder [1]
  (node2 :: Tensor Value Float) <- placeholder [1]
  let adderNode = node1 `add` node2
  let runStep = \node1Feed node2Feed -> runWithFeeds
        [ feed node1 node1Feed
        , feed node2 node2Feed
        ]
        adderNode
  runStep (encodeTensorData [1] input1) (encodeTensorData [1] input2)

main :: IO ()
main = do
  result1 <- runPlaceholder [3.0] [4.5]
  result2 <- runPlaceholder [2.7] [8.9]
  print result1
  print result2

{-
Output:
[7.5]
[11.599999]
-}
```

Now we'll wrap up by going through the simple linear model scenario we already saw in Python. Once again, we'll take two vectors as our inputs. These will be the values we try to match. Next, we'll use the initializedVariable function to get our variables. We don't need to call a global variable initializer. But this does affect the state of the session. Notice that we pull it out of the monad context, rather than using let. (We also did for placeholders.)

```
import TensorFlow.Core (Tensor, Value, feed, encodeTensorData, Scalar(..))
import TensorFlow.Ops (constant, 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)

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

runVariable :: Vector Float -> Vector Float -> IO (Float, Float)
runVariable xInput yInput = runSession $ do
  let xSize = fromIntegral $ Vector.length xInput
  let ySize = fromIntegral $ Vector.length yInput
  (w :: Variable Float) <- initializedVariable 3
  (b :: Variable Float) <- initializedVariable 1
  ...
```

Next, we'll make our placeholders and linear model. Then we'll calculate our loss function in much the same way we did before. Then we'll use the same feed trick to get our placeholders plugged in.

```
runVariable :: Vector Float -> Vector Float -> IO (Float, Float)
  ...
  (x :: Tensor Value Float) <- placeholder [xSize]
  let linear_model = ((readValue w) `mul` x) `add` (readValue b)
  (y :: Tensor Value Float) <- placeholder [ySize]
  let square_deltas = square (linear_model `sub` y)
  let loss = reduceSum square_deltas
  trainStep <- minimizeWith (gradientDescent 0.01) loss [w,b]
  let trainWithFeeds = \xF yF -> runWithFeeds
        [ feed x xF
        , feed y yF
        ]
        trainStep
...
```

Finally, we'll run our training step 1000 times on our input data. Then we'll run our model one more time to pull out the values of our weights and bias. Then we're done!

```
runVariable :: Vector Float -> Vector Float -> IO (Float, Float)
...
  replicateM_ 1000
    (trainWithFeeds (encodeTensorData [xSize] xInput) (encodeTensorData [ySize] yInput))
  (Scalar w_learned, Scalar b_learned) <- run (readValue w, readValue b)
  return (w_learned, b_learned)

main :: IO ()
main = do
  results <- runVariable [1.0, 2.0, 3.0, 4.0] [4.0, 9.0, 14.0, 19.0]
  print results

{-
Output:
(4.9999948,-0.99998516)
-}
```

CONCLUSION

Hopefully this article gave you a taste of some of the possibilities of Tensor Flow in Haskell. We saw a quick introduction to the fundamentals of Tensor Flow. We saw three different kinds of tensors. We then saw code examples both in Python and in Haskell. Finally, we went over a very quick example of a simple linear model and saw how we could learn values to fit that model.

Now that we've got the basics down, we're going to spice things up a lot! In part 2 we'll explore the question of program safety. We'll see that our Haskell code is not necessarily any better than the Python code! But then we'll see how we can use some awesome dependent type techniques to change this!

If you want more details on running this Tensor Flow code yourself, you should check out Haskell Tensor Flow Guide! It will walk you through using the Tensor Flow library as a dependency and getting a basic model running!

Revision #1
Created 2022-03-11 06:26:51 UTC by gasick
Updated 2022-03-11 17:11:16 UTC by gasick