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

Grenade and Deep Learning

In part 2 and part 3 of this series, we explored some of the most complex topics in Haskell. We examined potential runtime failures that can occur when using Tensor Flow. These included mismatched dimensions and missing placeholders. In an ideal world, we would catch these issues at compile time instead. At its current stage, the Haskell Tensor Flow library doesn't support that. But we demonstrated that it is possible to do this by using dependent types.

If you want to explore coding with the Haskell Tensor Flow library yourself, make sure you download our Guide. It'll give you a lot of tips on what dependencies you need and how to install everything!

Now, I'm still very much of a novice at dependent types, so the solutions I presented were rather clunky. In this final part, I'll show a better example of this concept from a different library. The Grenade library uses dependent types everywhere. It allows us to build verifiably-valid neural networks with extreme concision. Since it's so easy to build a larger network, Grenade can be a powerful tool for deep learning! So let's dive in and see what it's all about! The code for this part is on the grenade branch of our Github repository.

SHAPES AND LAYERS

The first thing to learn with this library is the twin concepts of Shapes and Layers. Shapes are best compared to tensors from Tensor Flow, except that they exist at the type level. In Tensor Flow we could build tensors with arbitrary dimensions. Grenade currently only supports up to three dimensions. So the different shape types either start with D1, D2, or D3, depending on the dimensionality of the shape. Then each of these type constructors take a set of natural number parameters. So the following are all valid “Shape” types within Grenade:

```
D1 5
D2 4 12
```

```
D3 8 10 2
```

The first represents a vector with 5 elements. The second represents a matrix with 4 rows and 12 columns. And the third represents an 8x10x2 matrix (or tensor, if you like). The different numbers represent those values at the type level, not the term level. If this seems confusing, here's a good tutorial that goes into more depth about the basics of dependent types. The most important idea is that something of type D1 5 can only have 5 elements. A vector of 4 or 6 elements will not type-check.

So now that we know about shapes, let's examine layers. Layers describe relationships between our shapes. They encapsulate the transformations that happen on our data. The following are all valid layer types:

```
Relu
FullyConnected 10 20
Convolution 1 10 5 5 1 1
```

The layer Relu describes a layer that takes in data of any kind of shape and outputs the same shape. In between, it applies the relu activation function to the input data. Since it doesn't change the shape, it doesn't need any parameters.

A FullyConnected layer represents the canonical layer of a neural network. It has two parameters, one for the number of input neurons and one for the number of output neurons. In this case, the layer will take 10 inputs and produce 20 outputs.

A Convolution layer represents a 2D convolution for our neural network. This particular example has 1 input feature, 10 output features, uses a 5x5 patch size, and a 1x1 patch offset.

DESCRIBING A NETWORK

Now that we have a basic grasp on shapes and layers, we can see how they fit together to create a full network. A network type has two type parameters. The second parameter is a list of the shapes that our data takes at any given point throughout the network. The first parameter is a list of the layers representing the transformations on the data. So let's say we wanted to describe a very simple network. It will take 4 inputs and produce 10 outputs using a fully connected layer. Then it

will perform an Relu activation. This network looks like this:

```
type SimpleNetwork = Network
  '[FullyConnected 4 10, Relu]
  '[ 'D1 4, 'D1 10, 'D1 10]
```

The apostrophes in front of the lists and D1 terms indicated that these are promoted constructors. So they are types instead of terms. To “read” this type, we start with the first data format. We go to each successive data format by applying the transformation layer. So for instance we start with a 4-vector, and transform it into a 10-vector with a fully-connected layer. Then we transform that 10-vector into another 10-vector by applying relu. That's all there is to it! We could apply another FullyConnected layer onto this that will have 3 outputs like so:

```
type SimpleNetwork = Network
  '[FullyConnected 4 10, Relu, FullyConnected 10 3]
  '[ 'D1 4, 'D1 10, 'D1 10, `D1 3]
```

Let's look at the MNIST digit recognition problem to see a more complicated example. We'll start with a 28x28 image of data. Then we'll perform the convolution layer I mentioned above. This gives us a 3-dimensional tensor of size 24x24x10. Then we can perform 2x2 max pooling on this, resulting in a 12x12x10 tensor. Finally, we can apply an Relu layer, which keeps it at the same size:

```
type MNISTStart = MNISTStart
  '[Convolution 1 10 5 5 1 1, Pooling 2 2 2 2, Relu]
  '[D2 28 28, D3 24 24 10, D3 12 12 10, D3 12 12 10]
```

Here's what a full MNIST example might look like (per the README on the library's Github page):

```
type MNIST = Network
    '[ Convolution 1 10 5 5 1 1, Pooling 2 2 2 2, Relu
     , Convolution 10 16 5 5 1 1, Pooling 2 2 2 2, FlattenLayer, Relu
     , FullyConnected 256 80, Logit, FullyConnected 80 10, Logit]
    '[ 'D2 28 28, 'D3 24 24 10, 'D3 12 12 10, 'D3 12 12 10
     , 'D3 8 8 16, 'D3 4 4 16, 'D1 256, 'D1 256
     , 'D1 80, 'D1 80, 'D1 10, 'D1 10]
```

This is a much simpler and more concise description of our network than we can get in Tensor Flow! Let's examine the ways in which the library uses dependent types to its advantage.

THE MAGIC OF DEPENDENT TYPES

Describing our network as a type seems like a strange idea if you've never used dependent types before. But it gives us a couple great perks!

The first major win we get is that it is very easy to generate the starting values of our network. Since it has a specific type, we can let type inference guide us! We don't need any term level code that is specific to the shape of our network. All we need to do is attach the type signature and call randomNetwork!

```
randomSimple :: MonadRandom m => m SimpleNetwork
randomSimple = randomNetwork
```

This will give us all the initial values we need, so we can get going!

The second (and more important) win is that we can't build an invalid network! Suppose we try to take our simple network and somehow format it incorrectly. For instance, we could say that instead of the input shape being of size 4, it's of size 7:

```
type SimpleNetwork = Network
  '[FullyConnected 4 10, Relu, FullyConnected 10 3]
  '[ 'D1 7, 'D1 10, 'D1 10, `D1 3]
-- ^^ Notice this 7
```

This will result in a compile error, since there is a mismatch between the layers. The first layer expects an input of 4, but the first data format is of length 7!

```
Could not deduce (Layer (FullyConnected 4 10) ('D1 7) ('D1 10))
        arising from a use of 'randomNetwork'
      from the context: MonadRandom m
        bound by the type signature for:
                   randomSimple :: MonadRandom m => m SimpleNetwork
        at src/IrisGrenade.hs:29:1-48
```

In other words, it notices that the chain from D1 7 to D1 10 using a FullyConnected 4 10 layer is invalid. So it doesn't let us make this network. The same thing would happen if we made the layers themselves invalid. For instance, we could make the output and input of the two fully-connected layers not match up:

```
-- We changed the second to take 20 as the number of input elements.
type SimpleNetwork = Network
  '[FullyConnected 4 10, Relu, FullyConnected 20 3]
  '[ 'D1 4, 'D1 10, 'D1 20, 'D1 3]

...

{- /Users/jamesbowen/HTensor/src/IrisGrenade.hs:30:16: error:
    • Could not deduce (Layer (FullyConnected 20 3) ('D1 10) ('D1 3))
        arising from a use of 'randomNetwork'
      from the context: MonadRandom m
        bound by the type signature for:
                   randomSimple :: MonadRandom m => m SimpleNetwork
        at src/IrisGrenade.hs:29:1-48
-}
```

So Grenade makes our program much safer by providing compile time guarantees about our network's validity. Runtime errors due to dimensionality are impossible!

TRAINING THE NETWORK ON IRIS

Now let's do a quick run-through of how we actually train this neural network. We'll use the Iris data set. We'll use the following steps:

Write the network type and generate a random network from it Read our input data into a format that Grenade uses Write a function to run a training iteration. Run it!

1. WRITE THE NETWORK TYPE AND GENERATE NETWORK

So we've already done this first step for the most part. We'll adjust the names a little bit though. Note that I'll include the imports list as an appendix to the post. Also, the code is on the grenade branch of my Haskell Tensor Flow repository in IrisGrenade.hs!

```
type IrisNetwork = Network
  '[FullyConnected 4 10, Relu, FullyConnected 10 3]
  '[ 'D1 4, 'D1 10, 'D1 10, 'D1 3]

randomIris :: MonadRandom m => m IrisNetwork
randomIris = randomNetwork

runIris :: FilePath -> FilePath -> IO ()
runIris trainingFile testingFile = do
  initialNetwork <- randomIris
  ...
```

2. TAKE IN OUR INPUT DATA

The readIrisFromFile function will take care of getting our data into a vector format. Then we'll make a dependent type called IrisRow, which uses the S type. This S type is a container for a shape. We want our input data to use D1 4 for the 4 input features. Then our output data should use D1 3 for the three possible categories.

```
-- Dependent type on the dimensions of the row
type IrisRow = (S ('D1 4), S ('D1 3))
```

If we have malformed data, the types will not match up, so we'll need to return a Maybe to ensure this succeeds. Note that we normalize the data by dividing by 8. This puts all the data between 0 and 1 and makes for better training results. Here's how we parse the data:

```
parseRecord :: IrisRecord -> Maybe IrisRow
parseRecord record = case (input, output) of
  (Just i, Just o) -> Just (i, o)
  _ -> Nothing
  where
    input = fromStorable $ VS.fromList $ float2Double <$>
      [ field1 record / 8.0, field2 record / 8.0, field3 record / 8.0, field4 record / 8.0]
    output = oneHot (fromIntegral $ label record)
```

Then we incorporate these into our main function:

```
runIris :: FilePath -> FilePath -> IO ()
runIris trainingFile testingFile = do
  initialNetwork <- randomIris
  trainingRecords <- readIrisFromFile trainingFile
  testRecords <- readIrisFromFile testingFile

  let trainingData = mapMaybe parseRecord (V.toList trainingRecords)
  let testData = mapMaybe parseRecord (V.toList testRecords)

  -- Catch if any were parsed as Nothing
  if length trainingData /= length trainingRecords || length testData /= length testRecords
    then putStrLn "Hmmm there were some problems parsing the data"
    else ...
```

3. WRITE A FUNCTION TO TRAIN THE INPUT DATA

This is a multi-step process. First we'll establish our learning parameters. We'll also write a function that will allow us to call the train function on a particular row element:

```
learningParams :: LearningParameters
learningParams = LearningParameters 0.01 0.9 0.0005

-- Train the network!
trainRow :: LearningParameters -> IrisNetwork -> IrisRow -> IrisNetwork
```

```
trainRow lp network (input, output) = train lp network input output
```

Next we'll write two more helper functions that will help us test our results. The first will take the network and a test row. It will transform it into the predicted output and the actual output of the network. The second function will take these outputs and reverse the oneHot process to get the label out (0, 1, or 2).

```
-- Takes a test row, returns predicted output and actual output from the network.
testRow :: IrisNetwork -> IrisRow -> (S ('D1 3), S ('D1 3))
testRow net (rowInput, predictedOutput) = (predictedOutput, runNet net rowInput)

-- Goes from probability output vector to label
getLabels :: (S ('D1 3), S ('D1 3)) -> (Int, Int)
getLabels (S1D predictedLabel, S1D actualOutput) =
  (maxIndex (extract predictedLabel), maxIndex (extract actualOutput))
```

Finally we'll write a function that will take our training data, test data, the network, and an iteration number. It will return the newly trained network, and log some results about how we're doing. We'll first take only a sample of our training data and adjust our parameters so that learning gets slower. Then we'll train the network by folding over the sampled data.

```
run :: [IrisRow] -> [IrisRow] -> IrisNetwork -> Int -> IO IrisNetwork
run trainData testData network iterationNum = do
  sampledRecords <- V.toList <$> chooseRandomRecords (V.fromList trainData)
  -- Slowly drop the learning rate
  let revisedParams = learningParams
        { learningRate = learningRate learningParams * 0.99 ^ iterationNum}
  let newNetwork = foldl' (trainRow revisedParams) network sampledRecords
  ....
```

Then we'll wrap up the function by looking at our test data, and seeing how much we got right!

```
run :: [IrisRow] -> [IrisRow] -> IrisNetwork -> Int -> IO IrisNetwork
    run trainData testData network iterationNum = do
      sampledRecords <- V.toList <$> chooseRandomRecords (V.fromList trainData)
      -- Slowly drop the learning rate
      let revisedParams = learningParams
            { learningRate = learningRate learningParams * 0.99 ^ iterationNum}
```

```
    let newNetwork = foldl' (trainRow revisedParams) network sampledRecords
    let labelVectors = fmap (testRow newNetwork) testData
    let labelValues = fmap getLabels labelVectors
    let total = length labelValues
    let correctEntries = length $ filter ((==) <$> fst <*> snd) labelValues
    putStrLn $ "Iteration: " ++ show iterationNum
    putStrLn $ show correctEntries ++ " correct out of: " ++ show total
    return newNetwork
```

4. RUN IT!

We'll call this now from our main function, iterating 100 times, and we're done!

```
runIris :: FilePath -> FilePath -> IO ()
runIris trainingFile testingFile = do
 ...
  if length trainingData /= length trainingRecords || length testData /= length testRecords
    then putStrLn "Hmmm there were some problems parsing the data"
    else foldM_ (run trainingData testData) initialNetwork [1..100]
```

COMPARING TO TENSOR FLOW

So now that we've looked at a different library, we can consider how it stacks up against Tensor Flow. So first, the advantages. Grenade's main advantage is that it provides dependent type facilities. This means it is more difficult to write incorrect programs. The basic networks you build are guaranteed to have the correct dimensionality. Additionally, it does not use a “placeholders” system, so you can avoid those kinds of errors too. This means you're likely to have fewer runtime bugs using Grenade.

Concision is another major strong point. The training code got a bit involved when translating our data into Grenade's format. But it's no more complicated than Tensor Flow. When it comes down to

the exact definition of the network itself, we do this in only a few lines with Grenade. It's complicated to understand what those lines mean if you are new to dependent types. But after seeing a few simple examples you should be able to follow the general pattern.

Of course, none of this means that Tensor Flow is without its advantages. Tensor Flow has much better logging utilities. The Tensor Board application will then give you excellent visualizations of this data. It is somewhat more difficult to get intermediate log results with Grenade. There is not too much transparency (that I have found at least) into the inner values of the network. The network types are composable though. So it is possible to get intermediate steps of your operation. But if you break your network into different types and stitch them together, you will remove some of the concision of the network.

Also, Tensor Flow also has a much richer ecosystem of machine learning tools to access. Grenade is still limited to a subset of the most common machine learning layers, like convolution and max pooling. Tensor Flow's API allows approaches like support vector machines and linear models. So Tensor Flow offers you more options.

CONCLUSION

Grenade provides some truly awesome facilities for building a concise neural network. A Grenade program can demonstrate at compile time that the network is well formed. It also allows an incredibly concise way to define what layers your neural network has. It doesn't have the Google level support that Tensor Flow does. So it lacks many cool features like logging and visualizations. But it is quite a neat library for its scope.

This concludes our series on Haskell and machine learning! If you want to get started writing some code yourself, the best place to start would be our Haskell Tensor Flow Guide. It will walk you through a lot of the tricks and gotchas when first getting Tensor Flow to work on your system. You can also take a look at the Github repository and examine all the different code examples we used in this series.

APPENDIX: COMPILER EXTENSIONS AND IMPORTS

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE TupleSections #-}
{-# LANGUAGE GADTs #-}

import          Control.Monad (foldM_)
import          Control.Monad.Random (MonadRandom)
import          Control.Monad.IO.Class (liftIO)
import          Data.Foldable (foldl')
import          Data.Maybe (mapMaybe)
import qualified Data.Vector.Storable as VS
import qualified Data.Vector as V
import          GHC.Float (float2Double)
import          Grenade
import          Grenade.Core.LearningParameters (LearningParameters(..))
import          Grenade.Core.Shape (fromStorable)
import          Grenade.Utils.OneHot (oneHot)
import          Numeric.LinearAlgebra (maxIndex)
import          Numeric.LinearAlgebra.Static (extract)

import          Processing (IrisRecord(..), readIrisFromFile, chooseRandomRecords)
```

Revision #1
Created 11 March 2022 06:35:59 by gasick
Updated 11 March 2022 17:11:16 by gasick