Learning MNIST with Neural Networks with backprop library

Justin Le

The *backprop* library performs back-propagation over a *hetereogeneous* system of relationships. back-propagation is done automatically (as reverse-mode automatic differentiation), and you work with your values as if you were writing normal functions with them, with the help of lens[1].

Repository source is on github[2], and docs are on hackage[3].

If you're reading this as a literate haskell file, you should know that a rendered pdf version is available on github.[4]. If you are reading this as a pdf file, you should know that a literate haskell version that you can run[5] is also available on github!

The (extra) packages involved are:

- hmatrix
- lens
- mnist-idx
- mwc-random
- one-liner-instances
- split

```
{-# LANGUAGE BangPatterns                      #-}
{-# LANGUAGE DataKinds                         #-}
{-# LANGUAGE DeriveGeneric                     #-}
{-# LANGUAGE FlexibleContexts                  #-}
{-# LANGUAGE GADTs                             #-}
{-# LANGUAGE LambdaCase                        #-}
{-# LANGUAGE ScopedTypeVariables               #-}
{-# LANGUAGE TemplateHaskell                   #-}
{-# LANGUAGE TupleSections                     #-}
{-# LANGUAGE TypeApplications                  #-}
{-# LANGUAGE ViewPatterns                      #-}
{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}
{-# OPTIONS_GHC -fno-warn-orphans              #-}
{-# OPTIONS_GHC -fno-warn-unused-top-binds     #-}

import           Control.DeepSeq
import           Control.Exception
import           Control.Lens hiding ((<.>))
import           Control.Monad
import           Control.Monad.IO.Class
```

[1] http://hackage.haskell.org/package/lens
[2] https://github.com/mstksg/backprop
[3] http://hackage.haskell.org/package/backprop
[4] https://github.com/mstksg/backprop/blob/master/renders/backprop-mnist.pdf
[5] https://github.com/mstksg/backprop/blob/master/samples/backprop-mnist.lhs

```haskell
import           Control.Monad.Trans.Maybe
import           Control.Monad.Trans.State
import           Data.Bitraversable
import           Data.Foldable
import           Data.IDX
import           Data.List.Split
import           Data.Time.Clock
import           Data.Traversable
import           Data.Tuple
import           GHC.Generics                       (Generic)
import           GHC.TypeLits
import           Numeric.Backprop
import           Numeric.LinearAlgebra.Static
import           Numeric.OneLiner
import           Text.Printf
import qualified Data.Vector                        as V
import qualified Data.Vector.Generic                as VG
import qualified Data.Vector.Unboxed                as VU
import qualified Numeric.LinearAlgebra              as HM
import qualified System.Random.MWC                  as MWC
import qualified System.Random.MWC.Distributions    as MWC
```

Introduction

In this walkthrough, we'll be building a classifier for the *MNIST*[6] data set. This is meant to mirror the Tensorflow Tutorial[7] for beginners.

Essentially, we use a two-layer artificial neural network – or a series of matrix multiplications, differentiable function applications, and vector additions. We feed our input image to the ANN and then try to get a label from it. Training an ANN is a matter of finding the right matrices to multiply by, and the right vectors to add.

To do that, we train our network by treating our network's accuracy as a function `Network -> Error`. If we can find the gradient of the input network with respect to the error, we can perform gradient descent[8], and slowly make our network better and better.

Finding the gradient is usually complicated, but *backprop* makes it simpler:

1. Write a function to compute the error from the network
2. That's it!

Hooray! Once you do that, the library finds the gradient function *automatically*, without any further intervention!

Types

For the most part, we're going to be using the great *hmatrix*[9] library and its vector and matrix types. It offers a type `L m n` for $m \times n$ matrices, and a type `R n` for an n vector.

[6] http://yann.lecun.com/exdb/mnist/
[7] https://www.tensorflow.org/versions/r1.2/get_started/mnist/beginners
[8] https://en.wikipedia.org/wiki/Gradient_descent
[9] http://hackage.haskell.org/package/hmatrix

First things first: let's define our neural networks as simple containers of parameters (weight matrices and bias vectors).

First, a type for layers:

```
data Layer i o =
    Layer { _lWeights :: !(L o i)
          , _lBiases  :: !(R o)
          }
  deriving (Show, Generic)

instance NFData (Layer i o)
makeLenses ''Layer
```

And a type for a simple feed-forward network with two hidden layers:

```
data Network i h1 h2 o =
    Net { _nLayer1 :: !(Layer i  h1)
        , _nLayer2 :: !(Layer h1 h2)
        , _nLayer3 :: !(Layer h2 o)
        }
  deriving (Show, Generic)

instance NFData (Network i h1 h2 o)
makeLenses ''Network
```

These are pretty straightforward container types... pretty much exactly the type you'd make to represent these networks! Note that, following true Haskell form, we separate out logic from data. This should be all we need.

Instances

Things are much simplier if we had `Num` and `Fractional` instances for everything, so let's just go ahead and define that now, as well. Just a little bit of boilerplate, made easier using *one-liner-instances*[10] to auto-derive instances using Generics.

```
instance (KnownNat i, KnownNat o) => Num (Layer i o) where
    (+)         = gPlus
    (-)         = gMinus
    (*)         = gTimes
    negate      = gNegate
    abs         = gAbs
    signum      = gSignum
    fromInteger = gFromInteger

instance ( KnownNat i
         , KnownNat h1
         , KnownNat h2
         , KnownNat o
         ) => Num (Network i h1 h2 o) where
    (+)         = gPlus
    (-)         = gMinus
    (*)         = gTimes
    negate      = gNegate
```

[10] http://hackage.haskell.org/package/one-liner-instances

```
    abs         = gAbs
    signum      = gSignum
    fromInteger = gFromInteger

instance (KnownNat i, KnownNat o) => Fractional (Layer i o) where
    (/)          = gDivide
    recip        = gRecip
    fromRational = gFromRational

instance ( KnownNat i
         , KnownNat h1
         , KnownNat h2
         , KnownNat o
         ) => Fractional (Network i h1 h2 o) where
    (/)          = gDivide
    recip        = gRecip
    fromRational = gFromRational
```

`KnownNat` comes from *base*; it's a typeclass that *hmatrix* uses to refer to the numbers in its type and use it to go about its normal hmatrixy business.

Ops

Now, *backprop* does require *primitive* differentiable operations on our relevant types to be defined. *backprop* uses these primitive operations to tie everything together. Ideally we'd import these from a library that implements these for you, and the end-user never has to make these primitives.

But in this case, I'm going to put the definitions here to show that there isn't any magic going on. If you're curious, refer to documentation for `Op`[11] for more details on how `Op` is implemented and how this works.

First, matrix-vector multiplication primitive, giving an explicit gradient function.

```
infixr 8 #>!
(#>!)
    :: (KnownNat m, KnownNat n, Reifies s W)
    => BVar s (L m n)
    -> BVar s (R n)
    -> BVar s (R m)
(#>!) = liftOp2 . op2 $ \m v ->
  ( m #> v, \g -> (g `outer` v, tr m #> g) )
```

Dot products would be nice too.

```
infixr 8 <.>!
(<.>!)
    :: (KnownNat n, Reifies s W)
    => BVar s (R n)
    -> BVar s (R n)
    -> BVar s Double
(<.>!) = liftOp2 . op2 $ \x y ->
  ( x <.> y, \g -> (konst g * y, x * konst g)
  )
```

[11] http://hackage.haskell.org/package/backprop/docs/Numeric-Backprop-Op.html

Also a function to fill a vector with the same element:

```
konst'
    :: (KnownNat n, Reifies s W)
    => BVar s Double
    -> BVar s (R n)
konst' = liftOp1 . op1 $ \c -> (konst c, HM.sumElements . extract)
```

Finally, an operation to sum all of the items in the vector.

```
sumElements'
    :: (KnownNat n, Reifies s W)
    => BVar s (R n)
    -> BVar s Double
sumElements' = liftOp1 . op1 $ \x -> (HM.sumElements (extract x), konst)
```

Again, these are not intended to be used by end-users of *backprop*, but rather are meant to be provided by libraries as primitive operations for users of the library to use.

Running our Network

Now that we have our primitives in place, let's actually write a function to run our network! And, once we do this, we automatically also have functions to back-propagate our network!

Normally, to write this function, we'd write:

```
runLayerNormal
    :: (KnownNat i, KnownNat o)
    => Layer i o
    -> R i
    -> R o
runLayerNormal l x = (l ^. lWeights) #> x + (l ^. lBiases)
{-# INLINE runLayerNormal #-}
```

Using the `lWeights` and `lBiases` lenses to access the weights and biases of our layer. However, we can translate this to *backprop* by operating on `BVar`s instead of the type directly, and using our backprop-aware `#>!`:

```
runLayer
    :: (KnownNat i, KnownNat o, Reifies s W)
    => BVar s (Layer i o)
    -> BVar s (R i)
    -> BVar s (R o)
runLayer l x = (l ^^. lWeights) #>! x + (l ^^. lBiases)
{-# INLINE runLayer #-}
```

`^.` lets to access data within a value using a lens, and `^^.` lets you access data within a `BVar` using a lens:

```
(^.)  ::        a -> Lens' a b ->        b
(^^.) :: BVar s a -> Lens' a b -> BVar s b
```

(There is also `^^?`, which can use a `Prism` or `Traversal` to extract a target that might not exist, `^^..`, which uses a `Traversal` to extract all targets, and `.~~`, which uses a `Lens` to update a value inside `BVar`)

Now `runLayer` is a function on two inputs that can be backpropagated, automatically! We can find its gradient given any input, and also run it to get our expected output as well.

Before writing our final network runner, we need a function to compute the "softmax" of our output vector. Writing it normally would look like:

```
softMaxNormal :: KnownNat n => R n -> R n
softMaxNormal x = konst (1 / HM.sumElements (extract expx)) * expx
  where
    expx = exp x
{-# INLINE softMaxNormal #-}
```

But we can make the mechanical shift to the backpropagatable version:

```
softMax :: (KnownNat n, Reifies s W) => BVar s (R n) -> BVar s (R n)
softMax x = konst' (1 / sumElements' expx) * expx
  where
    expx = exp x
{-# INLINE softMax #-}
```

We also need the logistic function[12], which is our activation function between layer outputs. Because `BVars` have a `Floating` instance, we can just write it using typeclass functions.

```
logistic :: Floating a => a -> a
logistic x = 1 / (1 + exp (-x))
{-# INLINE logistic #-}
```

With those in hand, let's compare how we would normally write a function to run our network:

```
runNetNormal
    :: (KnownNat i, KnownNat h1, KnownNat h2, KnownNat o)
    => Network i h1 h2 o
    -> R i
    -> R o
runNetNormal n = softMaxNormal
               . runLayerNormal (n ^. nLayer3)
               . logistic
               . runLayerNormal (n ^. nLayer2)
               . logistic
               . runLayerNormal (n ^. nLayer1)
{-# INLINE runNetNormal #-}
```

Basic function composition, neat. We use our lenses `nLayer1`, `nLayer2`, and `nLayer3` to extract the first, second, and third layers from our network.

Writing it in a way that backprop can use is also very similar:

```
runNetwork
    :: (KnownNat i, KnownNat h1, KnownNat h2, KnownNat o, Reifies s W)
    => BVar s (Network i h1 h2 o)
    -> R i
    -> BVar s (R o)
runNetwork n = softMax
             . runLayer (n ^^. nLayer3)
             . logistic
             . runLayer (n ^^. nLayer2)
             . logistic
             . runLayer (n ^^. nLayer1)
             . constVar
{-# INLINE runNetwork #-}
```

[12] https://en.wikipedia.org/wiki/Logistic_function

We use `constVar` on the input vector, because we don't care about its gradient and so treat it as a constant.

And now here again we use `^^.` (instead of `^.`) to extract a value from our `BVar` of a `Network`, using a lens.

Computing Errors

Now, training a neural network is about calculating its gradient with respect to some error function. The library calculatues the gradient for us – we just need to tell it how to compute the error function.

For classification problems, we usually use a cross entropy[13] error. Given a target vector, how does our neural network's output differ from what is expected? Lower numbers are better!

Again, let's look at a "normal" implementation, regular variables and no backprop:

```
crossEntropyNormal :: KnownNat n => R n -> R n -> Double
crossEntropyNormal targ res = -(log res <.> targ)
{-# INLINE crossEntropyNormal #-}
```

And we can see that the backpropable version is pretty similar. We see `constVar t`, to introduce a `BVar` that is a constant value (that we don't care about the gradient of).

```
crossEntropy
    :: (KnownNat n, Reifies s W)
    => R n
    -> BVar s (R n)
    -> BVar s Double
crossEntropy targ res = -(log res <.>! constVar targ)
{-# INLINE crossEntropy #-}
```

Our final "error function", then, is:

```
netErr
    :: (KnownNat i, KnownNat h1, KnownNat h2, KnownNat o, Reifies s W)
    => R i
    -> R o
    -> BVar s (Network i h1 h2 o)
    -> BVar s Double
netErr x targ n = crossEntropy targ (runNetwork n x)
{-# INLINE netErr #-}
```

The Magic

The actual "magic" of the library happens with the functions to "run" the functions we defined earlier:

```
evalBP   :: (forall s. Reifies s W => BVar s a -> BVar s b) -> a -> b
gradBP   :: (forall s. Reifies s W => BVar s a -> BVar s b) -> a -> a
backprop :: (forall s. Reifies s W => BVar s a -> BVar s b) -> a -> (b, a)
```

`evalBP` "runs" the function like normal, `gradBP` computes the gradient of the function, and `backprop` computes both the result and the gradient.

So, if we have a network `net0`, an input vector `x`, and a target vector `t`, we could compute its error using:

[13] https://en.wikipedia.org/wiki/Cross_entropy

```
evalBP (netErr x targ) net0 :: Double
```

And we can calculate its *gradient* using:

```
gradBP (netErr x targ) net0 :: (Network i h1 h2 o, R i)
```

Pulling it all together

Let's write a simple function to step our network in the direction opposite of the gradient to train our model:

```
trainStep
    :: forall i h1 h2 o. (KnownNat i, KnownNat h1, KnownNat h2, KnownNat o)
    => Double             -- ^ learning rate
    -> R i                -- ^ input
    -> R o                -- ^ target
    -> Network i h1 h2 o  -- ^ initial network
    -> Network i h1 h2 o
trainStep r !x !targ !n = n - realToFrac r * gradBP (netErr x targ) n
{-# INLINE trainStep #-}
```

Here's a convenient wrapper for training over all of the observations in a list:

```
trainList
    :: (KnownNat i, KnownNat h1, KnownNat h2, KnownNat o)
    => Double             -- ^ learning rate
    -> [(R i, R o)]       -- ^ input and target pairs
    -> Network i h1 h2 o  -- ^ initial network
    -> Network i h1 h2 o
trainList r = flip $ foldl' (\n (x,y) -> trainStep r x y n)
{-# INLINE trainList #-}
```

`testNet` will be a quick way to test our net by computing the percentage of correct guesses: (mostly using *hmatrix* stuff, so don't mind too much)

```
testNet
    :: forall i h1 h2 o. (KnownNat i, KnownNat h1, KnownNat h2, KnownNat o)
    => [(R i, R o)]
    -> Network i h1 h2 o
    -> Double
testNet xs n = sum (map (uncurry test) xs) / fromIntegral (length xs)
  where
    test :: R i -> R o -> Double          -- test if the max index is correct
    test x (extract->t)
        | HM.maxIndex t == HM.maxIndex (extract r) = 1
        | otherwise                                = 0
      where
        r :: R o
        r = evalBP (`runNetwork` x) n
```

And now, a main loop!

If you are following along at home, download the mnist data set files[14] and uncompress them into the folder `data`, and everything should work fine.

[14] http://yann.lecun.com/exdb/mnist/

```haskell
main :: IO ()
main = MWC.withSystemRandom $ \g -> do
    Just train <- loadMNIST "data/train-images-idx3-ubyte" "data/train-labels-idx1-ubyte"
    Just test  <- loadMNIST "data/t10k-images-idx3-ubyte"  "data/t10k-labels-idx1-ubyte"
    putStrLn "Loaded data."
    net0 <- MWC.uniformR @(Network 784 300 100 10) (-0.5, 0.5) g
    flip evalStateT net0 . forM_ [1..] $ \e -> do
      train' <- liftIO . fmap V.toList $ MWC.uniformShuffle (V.fromList train) g
      liftIO $ printf "[Epoch %d]\n" (e :: Int)

      forM_ ([1..] `zip` chunksOf batch train') $ \(b, chnk) -> StateT $ \n0 -> do
        printf "(Batch %d)\n" (b :: Int)

        t0 <- getCurrentTime
        n' <- evaluate . force $ trainList rate chnk n0
        t1 <- getCurrentTime
        printf "Trained on %d points in %s.\n" batch (show (t1 `diffUTCTime` t0))

        let trainScore = testNet chnk n'
            testScore  = testNet test n'
        printf "Training error:   %.2f%%\n" ((1 - trainScore) * 100)
        printf "Validation error: %.2f%%\n" ((1 - testScore ) * 100)

        return ((), n')
  where
    rate  = 0.02
    batch = 5000
```

Each iteration of the loop:

1. Shuffles the training set
2. Splits it into chunks of `batch` size
3. Uses `trainList` to train over the batch
4. Computes the score based on `testNet` based on the training set and the test set
5. Prints out the results

And, that's really it!

Performance

Currently, benchmarks show that *running* the network has virtually zero overhead (~ 4%) over writing the running function directly. The actual gradient descent process (compute gradient, then descend) carries about 60% overhead over writing the gradients manually, but it is unclear how much of this is because of the library, and how much of it is just because of automatic differentation giving slightly less efficient matrix/vector multiplication operations.

The README[15] has some more detailed benchmarks and statistics, if you want to get more detailed information.

[15] https://github.com/mstksg/backprop

Main takeaways

Most of the actual heavy lifting/logic actually came from the *hmatrix* library itself. We just created simple types to wrap up our bare matrices.

Basically, all that *backprop* did was give you an API to define *how to run* a neural net — how to *run* a net based on a `Network` and `R i` input you were given. The goal of the library is to let you write down how to run things in as natural way as possible.

And then, after things are run, we can just get the gradient and roll from there!

Because the heavy lifting is done by the data types themselves, we can presumably plug in *any* type and any tensor/numerical backend, and reap the benefits of those libraries' optimizations and parallelizations. *Any* type can be backpropagated! :D

What now?

Check out the docs for the Numeric.Backprop[16] module for a more detailed picture of what's going on, or find more examples at the github repo[17]!

Also, check out follow-up writeup to this tutorial, expanding on using the library with more advanced extensible neural network types, like the ones described in this blog post[18]. Check out the literate haskell here[19], and the rendered PDF here[20].

Boring stuff

Here is a small wrapper function over the mnist-idx[21] library loading the contents of the idx files into *hmatrix* vectors:

```
loadMNIST
    :: FilePath
    -> FilePath
    -> IO (Maybe [(R 784, R 10)])
loadMNIST fpI fpL = runMaybeT $ do
    i <- MaybeT          $ decodeIDXFile       fpI
    l <- MaybeT          $ decodeIDXLabelsFile fpL
    d <- MaybeT . return $ labeledIntData l i
    r <- MaybeT . return $ for d (bitraverse mkImage mkLabel . swap)
    liftIO . evaluate $ force r
  where
    mkImage :: VU.Vector Int -> Maybe (R 784)
    mkImage = create . VG.convert . VG.map (\i -> fromIntegral i / 255)
    mkLabel :: Int -> Maybe (R 10)
    mkLabel n = create $ HM.build 10 (\i -> if round i == n then 1 else 0)
```

And here are instances to generating random vectors/matrices/layers/networks, used for the initialization step.

[16] http://hackage.haskell.org/package/backprop/docs/Numeric-Backprop.html
[17] https://github.com/mstksg/backprop
[18] https://blog.jle.im/entries/series/+practical-dependent-types-in-haskell.html
[19] https://github.com/mstksg/backprop/blob/master/samples/extensible-neural.lhs
[20] https://github.com/mstksg/backprop/blob/master/renders/extensible-neural.pdf
[21] http://hackage.haskell.org/package/mnist-idx

```haskell
instance KnownNat n => MWC.Variate (R n) where
    uniform g = randomVector <$> MWC.uniform g <*> pure Uniform
    uniformR (l, h) g = (\x -> x * (h - l) + l) <$> MWC.uniform g

instance (KnownNat m, KnownNat n) => MWC.Variate (L m n) where
    uniform g = uniformSample <$> MWC.uniform g <*> pure 0 <*> pure 1
    uniformR (l, h) g = (\x -> x * (h - l) + l) <$> MWC.uniform g

instance (KnownNat i, KnownNat o) => MWC.Variate (Layer i o) where
    uniform g = Layer <$> MWC.uniform g <*> MWC.uniform g
    uniformR (l, h) g = (\x -> x * (h - l) + l) <$> MWC.uniform g

instance ( KnownNat i
         , KnownNat h1
         , KnownNat h2
         , KnownNat o
         )
      => MWC.Variate (Network i h1 h2 o) where
    uniform g = Net <$> MWC.uniform g <*> MWC.uniform g <*> MWC.uniform g
    uniformR (l, h) g = (\x -> x * (h - l) + l) <$> MWC.uniform g
```