Learning MNIST with Neural Networks with backprop library

Justin Le

The *backprop* library performs backpropagation over a *hetereogeneous* system of relationships. It offers both an implicit (ad[1]-like) and explicit graph building usage style. Let's use it to build neural networks and learn mnist!

Repository source is on github[2], and so are the rendered docs[3].

```haskell
{-# LANGUAGE BangPatterns         #-}
{-# LANGUAGE DataKinds            #-}
{-# LANGUAGE DeriveGeneric        #-}
{-# LANGUAGE GADTs                #-}
{-# LANGUAGE LambdaCase           #-}
{-# LANGUAGE ScopedTypeVariables  #-}
{-# LANGUAGE TupleSections        #-}
{-# LANGUAGE TypeApplications     #-}
{-# LANGUAGE ViewPatterns         #-}
{-# OPTIONS_GHC -fno-warn-orphans #-}

import           Control.DeepSeq
import           Control.Exception
import           Control.Monad
import           Control.Monad.IO.Class
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.Maybe
import           Data.Proxy
import           Data.Time.Clock
import           Data.Traversable
import           Data.Tuple
import           GHC.Generics                  (Generic)
import           GHC.TypeLits
import           Numeric.Backprop
import           Numeric.LinearAlgebra.Static hiding (dot)
import           Text.Printf
import qualified Data.Vector                   as V
import qualified Data.Vector.Generic           as VG
```

[1] http://hackage.haskell.org/package/ad
[2] https://github.com/mstksg/backprop
[3] https://mstksg.github.io/backprop

```haskell
import qualified Data.Vector.Unboxed                as VU
import qualified Generics.SOP                       as SOP
import qualified Numeric.LinearAlgebra              as LA
import qualified System.Random.MWC                  as MWC
import qualified System.Random.MWC.Distributions    as MWC
```

Types

For the most part, we're going to be using the great hmatrix[4] 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.

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

First, a type for layers:

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

instance SOP.Generic (Layer i o)
instance NFData (Layer i o)
```

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

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

instance SOP.Generic (Network i h1 h2 o)
instance NFData (Network i h1 h2 o)
```

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.

```haskell
instance (KnownNat i, KnownNat o) => Num (Layer i o) where
    Layer w1 b1 + Layer w2 b2 = Layer (w1 + w2) (b1 + b2)
    Layer w1 b1 - Layer w2 b2 = Layer (w1 - w2) (b1 - b2)
    Layer w1 b1 * Layer w2 b2 = Layer (w1 * w2) (b1 * b2)
    abs    (Layer w b)        = Layer (abs w) (abs b)
    signum (Layer w b)        = Layer (signum w) (signum b)
```

[4] http://hackage.haskell.org/package/hmatrix

```haskell
    negate (Layer w b)         = Layer (negate w) (negate b)
    fromInteger x              = Layer (fromInteger x) (fromInteger x)

instance (KnownNat i, KnownNat h1, KnownNat h2, KnownNat o) => Num (Network i h1 h2 o) wher
    Net a b c + Net d e f = Net (a + d) (b + e) (c + f)
    Net a b c - Net d e f = Net (a - d) (b - e) (c - f)
    Net a b c * Net d e f = Net (a * d) (b * e) (c * f)
    negate (Net a b c)    = Net (negate a) (negate b) (negate c)
    signum (Net a b c)    = Net (signum a) (signum b) (signum c)
    abs    (Net a b c)    = Net (abs    a) (abs    b) (abs    c)
    fromInteger x         = Net (fromInteger x) (fromInteger x) (fromInteger x)

instance (KnownNat i, KnownNat o) => Fractional (Layer i o) where
    Layer w1 b1 / Layer w2 b2 = Layer (w1 / w2) (b1 / b2)
    recip (Layer w b)          = Layer (recip w) (recip b)
    fromRational x             = Layer (fromRational x) (fromRational x)

instance (KnownNat i, KnownNat h1, KnownNat h2, KnownNat o) => Fractional (Network i h1 h2
    Net a b c / Net d e f = Net (a / d) (b / e) (c / f)
    recip (Net a b c)     = Net (recip a) (recip b) (recip c)
    fromRational x        = Net (fromRational x) (fromRational x) (fromRational x)
```

`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 `Op`s 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 `Op` primitives.

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

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

```haskell
matVec
    :: (KnownNat m, KnownNat n)
    => Op '[ L m n, R n ] (R m)
matVec = op2' $ \m v -> ( m #> v
                        , \(fromMaybe 1 -> g) ->
                             (g `outer` v, tr m #> g)
                        )
```

Dot products would be nice too.

```haskell
dot :: KnownNat n
    => Op '[ R n, R n ] Double
dot = op2' $ \x y -> ( x <.> y
                     , \case Nothing -> (y, x)
                             Just g  -> (konst g * y, x * konst g)
                     )
```

Also a "constant vector" function, which generates a constant vector from a given element.

```
rkonst
    :: forall n. KnownNat n
    => Op '[ Double ] (R n)
rkonst = op1' $ \x -> (konst x, maybe (fromIntegral (natVal @n Proxy))
                                      (LA.sumElements . extract)
                      )
```

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

```
rsum
    :: KnownNat n
    => Op '[ R n ] Double
rsum = op1' $ \x -> (LA.sumElements (extract x), maybe 1 konst)
```

And why not, here's a logistic function. We don't need to define this as an `Op` up-front, because the library can automatically promote any numeric polymorphic `a -> a` or `a -> a -> a` to an `Op` anyways later.

```
logistic :: Floating a => a -> a
logistic x = 1 / (1 + exp (-x))
```

Running our Network

Now that we have our primitives in place, let's actually write a function to run our network!

```
runLayer
    :: (KnownNat i, KnownNat o)
    => BPOp s '[ R i, Layer i o ] (R o)
runLayer = withInps $ \(x :< l :< Ø) -> do
    w :< b :< Ø <- gTuple #<~ l
    y <- matVec ~$ (w :< x :< Ø)
    return $ y + b
```

A `BPOp s '[ R i, Layer i o ] (R o)` is a backpropagatable function that produces an `R o` (a vector with `o` elements, from the hmatrix[5] library) given an input environment of an `R i` (the "input" of the layer) and a layer.

We use `withInps` to bring the environment into scope as a bunch of `BVar`s. `x` is a `BVar` containing to the input vector, and `l` is a `BVar` containing to the layer.

The first thing we do is split out the parts of the layer so we can work with the internal matrices. We can use `#<~` to "split out" the components of a `BVar`, splitting on `gTuple` (which uses `GHC.Generics` to automatically figure out how to split up a product type).

Then we apply `matVec` (our primitive `Op` that does matrix-vector multiplication) to `w` and `x`, and then the result is that added to the bias vector `b`.

We can write the `runNetwork` function pretty much the same way.

```
runNetwork
    :: (KnownNat i, KnownNat h1, KnownNat h2, KnownNat o)
    => BPOp s '[ R i, Network i h1 h2 o ] (R o)
runNetwork = withInps $ \(x :< n :< Ø) -> do
    l1 :< l2 :< l3 :< Ø <- gTuple #<~ n
    y <- runLayer -$ (x          :< l1 :< Ø)
    z <- runLayer -$ (logistic y :< l2 :< Ø)
```

[5] http://hackage.haskell.org/package/hmatrix

```haskell
    r <- runLayer -$ (logistic z :< l3 :< Ø)
    softmax        -$ (r            :< Ø)
  where
    softmax :: KnownNat n => BPOp s '[ R n ] (R n)
    softmax = withInps $ \(x :< Ø) -> do
        expX <- bindVar (exp x)
        totX <- rsum ~$ (expX :< Ø)
        return $ expX / liftB1 rkonst totX
```

After splitting out the layers in the input `Network`, we run each layer successively using our previously defined `runLayer`, giving inputs using `-$`. We can directly apply `logistic` to `BVar`s. At the end, we run a softmax function[6] because MNIST is a classification challenge. The softmax is done by applying e^x for every item in the input vector, and dividing each element by the total.

The Magic

What did we just define? Well, with a `BPOp s rs a`, we can *run* it and get the output:

```haskell
runNetOnInp
    :: (KnownNat i, KnownNat h1, KnownNat h2, KnownNat o)
    => Network i h1 h2 o
    -> R i
    -> R o
runNetOnInp n x = evalBPOp runNetwork (x ::< n ::< Ø)
```

But, the magic part is that we can also get the gradient!

```haskell
gradNet
    :: (KnownNat i, KnownNat h1, KnownNat h2, KnownNat o)
    => Network i h1 h2 o
    -> R i
    -> Network i h1 h2 o
gradNet n x = case gradBPOp runNetwork (x ::< n ::< Ø) of
    gradX ::< gradN ::< Ø -> gradN
```

This gives the gradient of all of the parameters in the matrices and vectors inside the `Network`, which we can use to "train"!

Training

Now for the real work. To train a network, we can do gradient descent based on the gradient of some type of *error function* with respect to the network parameters. Let's use the cross entropy[7], which is popular for classification problems.

```haskell
crossEntropy
    :: KnownNat n
    => R n
    -> BPOpI s '[ R n ] Double
crossEntropy targ (r :< Ø) = negate (dot .$ (log r :< t :< Ø))
  where
    t = constVar targ
```

[6] https://en.wikipedia.org/wiki/Softmax_function
[7] https://en.wikipedia.org/wiki/Cross_entropy

Given a target vector and a `BVar` referring to the result of the network, we can directly apply:

$$H(\mathbf{r}, \mathbf{t}) = -(log(\mathbf{r}) \cdot \mathbf{t})$$

Just for fun, I implemented `crossEntropy` in "implicit-graph" mode, so you don't see any binds or returns.

Now, a function to make one gradient descent step based on an input vector and a target, using `gradBPOp`:

```
trainStep
    :: forall i h1 h2 o. (KnownNat i, KnownNat h1, KnownNat h2, KnownNat o)
    => Double
    -> R i
    -> R o
    -> Network i h1 h2 o
    -> Network i h1 h2 o
trainStep r !x !t !n = case gradBPOp o (x ::< n ::< Ø) of
    _ ::< gN ::< Ø ->
        n - (realToFrac r * gN)
  where
    o :: BPOp s '[ R i, Network i h1 h2 o ] Double
    o = do
      y <- runNetwork
      implicitly (crossEntropy t) -$ (y :< Ø)
```

A convenient wrapper for training over all of the observations in a list:

```
trainList
    :: (KnownNat i, KnownNat h1, KnownNat h2, KnownNat o)
    => Double
    -> [(R i, R o)]
    -> Network i h1 h2 o
    -> Network i h1 h2 o
trainList r = flip $ foldl' (\n (x,y) -> trainStep r x y n)
```

Pulling it all together

`testNet` will be a quick way to test our net by computing the percentage of correct guesses: (mostly using *hmatrix* stuff)

```
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 x (extract->t)
        | LA.maxIndex t == LA.maxIndex (extract r) = 1
        | otherwise                                = 0
      where
        r :: R o
        r = evalBPOp runNetwork (x ::< n ::< Ø)
```

And now, a main loop!

If you are following along at home, download the mnist data set files[8] and uncompress them into the folder `data`, and everything should work fine.

```
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 9) (-1, 1) 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  = 0.1
    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!

Result

I haven't put much into optimizing the library yet, but the network (with hidden layer sizes 300 and 100) seems to take 30s on my computer to finish a batch of 5000 points. It's slow, but it's a first unoptimized run and a proof of concept! It's my goal to get this down to a point where the result has the same performance characteristics as the actual backend (*hmatrix*), and so overhead is 0.

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

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.

Boring stuff

Here is a small wrapper function over the mnist-idx[9] library loading the contents of the idx files into *hmatrix* vectors:

```
loadMNIST
    :: FilePath
    -> FilePath
    -> IO (Maybe [(R 784, R 9)])
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 9)
    mkLabel n = create $ LA.build 9 (\i -> if i == fromIntegral n then 1 else 0)
```

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

```
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
    uniform g = Net <$> MWC.uniform g <*> MWC.uniform g <*> MWC.uniform g
    uniformR (l, h) g = (\x -> x * (h - l) + l) <$> MWC.uniform g
```

[9] http://hackage.haskell.org/package/mnist-idx