

Extensible Neural Networks with Backprop

Justin Le

This write-up is a follow-up to the *MNIST* tutorial (rendered¹ here, and literate haskell² here). This write-up itself is available as a literate haskell file³, and also rendered as a pdf⁴.

The (extra) packages involved are:

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

```
{-# LANGUAGE BangPatterns      #-}
{-# LANGUAGE DataKinds         #-}
{-# LANGUAGE DeriveGeneric     #-}
{-# LANGUAGE FlexibleContexts   #-}
{-# LANGUAGE GADTs             #-}
{-# LANGUAGE InstanceSigs      #-}
{-# LANGUAGE LambdaCase        #-}
{-# LANGUAGE LambdaCase        #-}
{-# LANGUAGE RankNTypes        #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TemplateHaskell    #-}
{-# LANGUAGE TypeApplications  #-}
{-# LANGUAGE TypeInType        #-}
{-# LANGUAGE TypeOperators      #-}
{-# LANGUAGE ViewPatterns       #-}
{-# OPTIONS_GHC -fno-warn-orphans #-}

import Control.DeepSeq
import Control.Exception
import Control.Lens hiding      ((<.>))
import Control.Monad
import Control.Monad.IO.Class
import Control.Monad.Primitive
import Control.Monad.Trans.Maybe
import Control.Monad.Trans.State
import Data.Bitraversable
import Data.Foldable
import Data.IDX
```

¹<https://github.com/mstksg/backprop/blob/master/renderers/backprop-mnist.pdf>

²<https://github.com/mstksg/backprop/blob/master/samples/backprop-mnist.lhs>

³<https://github.com/mstksg/backprop/blob/master/samples/extensible-neural.lhs>

⁴<https://github.com/mstksg/backprop/blob/master/renderers/extensible-neural.pdf>

```

import Data.Kind
import Data.List.Split
import Data.Singletons
import Data.Singletons.Prelude
import Data.Singletons.TypeLits
import Data.Time.Clock
import Data.Traversable
import Data.Tuple
import GHC.Generics (Generic)
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

The *backprop*⁵ library lets us manipulate our values in a natural way. We write the function to compute our result, and the library then automatically finds the *gradient* of that function, which we can use for gradient descent.

In the last post, we looked at using a fixed-structure neural network. However, in this blog series⁶, I discuss a system of extensible neural networks that can be chained and composed.

One issue, however, in naively translating the implementations, is that we normally run the network by pattern matching on each layer. However, we cannot directly pattern match on `BVars`.

We *could* get around it by being smart with prisms and `^^?`, to extract a “Maybe BVar”. However, we can do better! This is because the *shape* of a `Net i h s o` is known already at compile-time, so there is no need for runtime checks like prisms and `^^?`.

Instead, we can just directly use lenses, since we know *exactly* what constructor will be present! We can use singletons to determine which constructor is present, and so always just directly use lenses without any runtime nondeterminism.

Types

First, our types:

```

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

```

⁵<http://hackage.haskell.org/package/backprop>

⁶<https://blog.jle.im/entries/series/+practical-dependent-types-in-haskell.html>

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

data Net :: Nat -> [Nat] -> Nat -> Type where
  NO    :: !(Layer i o) -> Net i '[] o
  (:~)  :: !(Layer i h) -> !(Net h hs o) -> Net i (h ': hs) o
```

Unfortunately, we can't automatically generate lenses for GADTs, so we have to make them by hand.^[poly] with type safety via parametric polymorphism.

```
_NO :: Lens (Net i '[] o) (Net i' '[] o')
      (Layer i o) (Layer i' o')
_NO f (NO l) = NO <$> f l

_NIL :: Lens (Net i (h ': hs) o) (Net i' (h ': hs) o)
      (Layer i h) (Layer i' h)
_NIL f (l :~ n) = (:~ n) <$> f l

_NIN :: Lens (Net i (h ': hs) o) (Net i' (h ': hs') o')
      (Net h hs o) (Net h hs' o')
_NIN f (l :~ n) = (l :~) <$> f n
```

You can read `_NO` as:

```
_NO :: Lens' (Net i '[] o) (Layer i o)
```

A lens into a single-layer network, and

```
_NIL :: Lens' (Net i (h ': hs) o) (Layer i h)
_NIN :: Lens' (Net i (h ': hs) o) (Net h hs o)
```

Lenses into a multiple-layer network, getting the first layer and the tail of the network.

If we pattern match on `Sing hs`, we can always determine exactly which lenses we can use, and so never fumble around with prisms or nondeterminism.

Running the network

Here's the meat of process, then: specifying how to run the network. We re-use our `BVar`-based combinators defined in the last write-up:

```
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 #-}
```

For `runNetwork`, we pattern match on `hs` using singletons, so we always know exactly what type of network we have:

```
runNetwork
  :: (KnownNat i, KnownNat o, Reifies s W)
  => BVar s (Net i hs o)
  -> Sing hs
```

```

-> BVar s (R i)
-> BVar s (R o)
runNetwork n = \case
  SNil          -> softmax . runLayer (n ^^. _NO)
  SCons SNat hs -> withSingI hs (runNetwork (n ^^. _NIN) hs)
                  . logistic
                  . runLayer (n ^^. _NIL)
{-# INLINE runNetwork #-}

```

The rest of it is the same as before.

```

netErr
  :: (KnownNat i, KnownNat o, SingI hs, Reifies s W)
  => R i
  -> R o
  -> BVar s (Net i hs o)
  -> BVar s Double
netErr x targ n = crossEntropy targ (runNetwork n sing (constVar x))
{-# INLINE netErr #-}

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

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

testNet
  :: forall i hs o. (KnownNat i, KnownNat o, SingI hs)
  => [(R i, R o)]
  -> Net i hs 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 (\n' -> runNetwork n' sing (constVar x)) n

```

And that's it!

Running

Everything here is the same as before, except now we can dynamically pick the network size. Here we pick `'[300,100]` for the hidden layer sizes.

```
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 @(Net 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
```

Looking Forward

One common thing people might do is want to be able to mix different types of layers. This could also be easily encoded as different constructors in `Layer`, and so `runLayer` will now be different depending on what constructor is present.

In this case, we can either:

1. Have a different indexed type for layers, so that we can always know exactly what layer is involved, so we don't have to runtime pattern match:

```
data LayerType = FullyConnected | Convolutional

data Layer :: LayerType -> Nat -> Nat -> Type where
  LayerFC :: .... -> Layer 'FullyConnected i o
  LayerC  :: .... -> Layer 'Convolutional i o
```

We would then have `runLayer` take `Sing (t :: LayerType)`, so we can again use `^^.` and directly pattern match.

2. Use a typeclass-based approach, so users can add their own layer types. In this situation, layer types

would all be different types, and running them would be a typeclass method that would give our `BVar s (Layer i o) -> BVar s (R i) -> BVar s (R o)` operation as a typeclass method.

```
class Layer (l :: Nat -> Nat -> Type) where
  runLayer
    :: forall s. Reifies s W
    => BVar s (l i o)
    -> BVar s (R i)
    -> BVar s (R o)
```

In all cases, it shouldn't be much more cognitive overhead to use *backprop* to build your neural network framework!

And, remember that `evalBP` (directly running the function) introduces virtually zero overhead, so if you only provided `BVar` functions, you could easily get the original non-`BVar` functions with `evalBP` without any loss.

What now?

Ready to start? Check out the docs for the `Numeric.Backprop`⁷ module for the full technical specs, and find more examples and updates at the [github repo](https://github.com/mstksg/backprop)⁸!

Internals

That's it for the post! Now for the internal plumbing :)

```
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)
```

HMatrix Operations

```
infixr 8 #>!
(#>!)
  :: (KnownNat m, KnownNat n, Reifies s W)
  => BVar s (L m n)
  -> BVar s (R n)
```

⁷<http://hackage.haskell.org/package/backprop/docs/Numeric-Backprop.html>

⁸<https://github.com/mstksg/backprop>

```

    -> BVar s (R m)
  (#>!) = liftOp2 . op2 $ \m v ->
    ( m #> v, \g -> (g `outer` v, tr m #> g) )

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)
    )

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

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

  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 #-}

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

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

```

Instances

```

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

liftNet0
  :: forall i hs o. (KnownNat i, KnownNat o)
  => (forall m n. (KnownNat m, KnownNat n) => Layer m n)
  -> Sing hs
  -> Net i hs o
liftNet0 x = go
where
  go :: forall w ws. KnownNat w => Sing ws -> Net w ws o
  go = \case
    SNil          -> NO x
    SCons SNat hs -> x :~ go hs

liftNet1
  :: forall i hs o. (KnownNat i, KnownNat o)
  => (forall m n. (KnownNat m, KnownNat n)
    => Layer m n
    -> Layer m n
    )
  -> Sing hs
  -> Net i hs o
  -> Net i hs o
liftNet1 f = go
where
  go :: forall w ws. KnownNat w
    => Sing ws
    -> Net w ws o
    -> Net w ws o
  go = \case
    SNil          -> \case
      NO x -> NO (f x)
    SCons SNat hs -> \case
      x :~ xs -> f x :~ go hs xs

liftNet2
  :: forall i hs o. (KnownNat i, KnownNat o)
  => (forall m n. (KnownNat m, KnownNat n)
    => Layer m n
    -> Layer m n
    -> Layer m n
    )
  -> Sing hs
  -> Net i hs o
  -> Net i hs o
  -> Net i hs o

```



```

liftNet2 f = go
  where
    go :: forall w ws. KnownNat w
      => Sing ws
      -> Net w ws o
      -> Net w ws o
      -> Net w ws o
    go = \case
      SNil          -> \case
        NO x -> \case
          NO y -> NO (f x y)
        SCons SNat hs -> \case
          x :~ xs -> \case
            y :~ ys -> f x y :~ go hs xs ys

instance ( KnownNat i
          , KnownNat o
          , SingI hs
          )
  => Num (Net i hs o) where
  (+)      = liftNet2 (+) sing
  (-)      = liftNet2 (-) sing
  (*)      = liftNet2 (*) sing
  negate   = liftNet1 negate sing
  abs      = liftNet1 abs sing
  signum   = liftNet1 signum sing
  fromInteger x = liftNet0 (fromInteger x) sing

instance ( KnownNat i
          , KnownNat o
          , SingI hs
          )
  => Fractional (Net i hs o) where
  (/)      = liftNet2 (/) sing
  recip    = liftNet1 negate sing
  fromRational x = liftNet0 (fromRational x) sing

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 o
          , SingI hs
          )

```

```

=> MWC.Variate (Net i hs o) where
uniform :: forall m. PrimMonad m => MWC.Gen (PrimState m) -> m (Net i hs o)
uniform g = go sing
where
  go :: forall w ws. KnownNat w => Sing ws -> m (Net w ws o)
  go = \case
    SNil          -> NO <$> MWC.uniform g
    SCons SNat hs -> (:~) <$> MWC.uniform g <*> go hs
  uniformR (l, h) g = (\x -> x * (h - l) + l) <$> MWC.uniform g

instance NFData (Net i hs o) where
  rnf = \case
    NO l      -> rnf l
    x :~ xs -> rnf x `seq` rnf xs

```