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!

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

```
{-# LANGUAGE DeriveGeneric        #-}
{-# LANGUAGE GADTs                #-}
{-# LANGUAGE LambdaCase           #-}
{-# LANGUAGE RankNTypes           #-}
{-# LANGUAGE ScopedTypeVariables  #-}
{-# LANGUAGE StandaloneDeriving   #-}
{-# LANGUAGE TypeApplications     #-}
{-# LANGUAGE TypeInType           #-}
{-# LANGUAGE TypeOperators        #-}
{-# LANGUAGE ViewPatterns         #-}
{-# OPTIONS_GHC -fno-warn-orphans #-}

import           Data.Functor
import           Data.Kind
import           Data.Maybe
import           Data.Singletons
import           Data.Singletons.Prelude
import           Data.Singletons.TypeLits
import           Data.Type.Combinator
import           Data.Type.Product
import           GHC.Generics                        (Generic)
import           Numeric.Backprop
import           Numeric.Backprop.Iso
import           Numeric.LinearAlgebra.Static hiding (dot)
import           System.Random.MWC
import qualified Generics.SOP                       as SOP
```

Ops

First, we define values of `Op` for the operations we want to do. `Op`s are bundles of functions packaged with their hetereogeneous gradients. For simple numeric functions, *backprop* can derive `Op`s automatically. But for matrix operations, we have to derive them ourselves.

The types help us with matching up the dimensions, but we still need to be careful that our gradients are calculated correctly.

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

`L` and `R` are matrix and vector types from the great *hmatrix* library.

First, matrix-vector multiplication:

```
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)
                        )
```

Now, dot products:

```
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)
                     )
```

Polymorphic functions can be easily turned into `Op`s with `op1`/`op2` etc., but they can also be run directly on graph nodes.

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

A Simple Complete Example

At this point, we already have enough to train a simple single-hidden-layer neural network:

```
simpleOp
      :: (KnownNat m, KnownNat n, KnownNat o)
      => R m
      -> BPOpI s '[ L n m, R n, L o n, R o ] (R o)
simpleOp inp = \(w1 :< b1 :< w2 :< b2 :< Ø) ->
    let z = logistic $ liftB2 matVec w1 x + b1
    in  logistic $ liftB2 matVec w2 z + b2
  where
    x = constVar inp
```

Here, `simpleOp` is defined in implicit (non-monadic) style, given a tuple of inputs and returning outputs. Now `simpleOp` can be "run" with the input vectors and parameters (a `L n m`, `R n`, `L o n`, and `R o`) and calculate the output of the neural net.

```
runSimple
    :: (KnownNat m, KnownNat n, KnownNat o)
    => R m
    -> Tuple '[ L n m, R n, L o n, R o ]
    -> R o
runSimple inp = evalBPOp (implicitly $ simpleOp inp)
```

Alternatively, we can define `simpleOp` in explicit monadic style, were we specify our graph nodes explicitly. The results should be the same.

```
simpleOpExplicit
      :: (KnownNat m, KnownNat n, KnownNat o)
```

```haskell
      => R m
      -> BPOp s '[ L n m, R n, L o n, R o ] (R o)
simpleOpExplicit inp = withInps $ \(w1 :< b1 :< w2 :< b2 :< Ø) -> do
    -- First layer
    y1  <- matVec ~$ (w1 :< x1 :< Ø)
    let x2 = logistic (y1 + b1)
    -- Second layer
    y2  <- matVec ~$ (w2 :< x2 :< Ø)
    return $ logistic (y2 + b2)
  where
    x1 = constVar inp
```

Now, for the magic of *backprop*: the library can now take advantage of the implicit (or explicit) graph and use it to do backpropagation, too!

```haskell
simpleGrad
    :: forall m n o. (KnownNat m, KnownNat n, KnownNat o)
    => R m
    -> R o
    -> Tuple '[ L n m, R n, L o n, R o ]
    -> Tuple '[ L n m, R n, L o n, R o ]
simpleGrad inp targ params = gradBPOp opError params
  where
    opError :: BPOp s '[ L n m, R n, L o n, R o ] Double
    opError = do
        res <- implicitly $ simpleOp inp
        -- we explicitly bind err to prevent recomputation
        err <- bindVar $ res - t
        dot ~$ (err :< err :< Ø)
      where
        t = constVar targ
```

The result is the gradient of the input tuple's components, with respect to the `Double` result of `opError` (the squared error). We can then use this gradient to do gradient descent.

With Parameter Containers

This method doesn't quite scale, because we might want to make networks with multiple layers and parameterize networks by layers. Let's make some basic container data types to help us organize our types, including a recursive `Network` type that lets us chain multiple layers.

```haskell
data Layer :: Nat -> Nat -> Type where
    Layer :: { _lWeights :: L m n
             , _lBiases  :: R m
             }
          -> Layer n m
      deriving (Show, Generic)


data Network :: Nat -> [Nat] -> Nat -> Type where
    NØ   :: !(Layer a b) -> Network a '[] b
    (:&) :: !(Layer a b) -> Network b bs c -> Network a (b ': bs) c
```

A `Layer n m` is a layer taking an n-vector and returning an m-vector. A `Network a '[b, c, d] e` would be a Network that takes in an a-vector and outputs an e-vector, with hidden layers of sizes b, c, and d.

Isomorphisms

The *backprop* library lets you apply operations on "parts" of data types (like on the weights and biases of a `Layer`) by using `Iso`'s (isomorphisms), like the ones from the *lens* library. The library doesn't depend on lens, but it can use the `Iso`s from the library and also custom-defined ones.

First, we can auto-generate isomorphisms using the *generics-sop* library:

```
instance SOP.Generic (Layer n m)
```

And then can create isomorphisms by hand for the two `Network` constructors:

```
netExternal :: Iso' (Network a '[] b) (Tuple '[Layer a b])
netExternal = iso (\case NØ x      -> x ::< Ø)
                  (\case I x :< Ø -> NØ x    )

netInternal :: Iso' (Network a (b ': bs) c) (Tuple '[Layer a b, Network b bs c])
netInternal = iso (\case x :& xs          -> x ::< xs ::< Ø)
                  (\case I x :< I xs :< Ø -> x :& xs        )
```

An `Iso' a (Tuple as)` means that an `a` can really just be seen as a tuple of `as`.

Running a network

Now, we can write the `BPOp` that reprenents running the network and getting a result. We pass in a `Sing bs` (a singleton list of the hidden layer sizes) so that we can "pattern match" on the list and handle the different network constructors differently.

```
netOp
    :: forall s a bs c. (KnownNat a, KnownNat c)
    => Sing bs
    -> BPOp s '[ R a, Network a bs c ] (R c)
netOp sbs = go sbs
  where
    go :: forall d es. KnownNat d
        => Sing es
        -> BPOp s '[ R d, Network d es c ] (R c)
    go = \case
      SNil -> withInps $ \(x :< n :< Ø) -> do
        -- peek into the NØ using netExternal iso
        l :< Ø <- netExternal #<~ n
        -- run the 'layerOp' BP, with x and l as inputs
        bpOp layerOp ~$ (x :< l :< Ø)
      SNat `SCons` ses -> withInps $ \(x :< n :< Ø) -> withSingI ses $ do
        -- peek into the (:&) using the netInternal iso
        l :< n' :< Ø <- netInternal #<~ n
        -- run the 'layerOp' BP, with x and l as inputs
        z <- bpOp layerOp  ~$ (x :< l :< Ø)
        -- run the 'go ses' BP, with z and n as inputs
        bpOp (go ses)      ~$ (z :< n' :< Ø)
    layerOp
        :: forall d e. (KnownNat d, KnownNat e)
```

```
        => BPOp s '[ R d, Layer d e ] (R e)
    layerOp = withInps $ \(x :< l :< Ø) -> do
        -- peek into the layer using the gTuple iso, auto-generated with SOP.Generic
        w :< b :< Ø <- gTuple #<~ l
        y           <- matVec  ~$ (w :< x :< Ø)
        return $ logistic (y + b)
```

There's some singletons work going on here, but it's fairly standard singletons stuff. Most of the complexity here is from the static typing in our neural network type, and *not* from *backprop*.

From *backprop* specifically, the only elements are `#<~` lets you "split" an input ref with the given iso, and `bpOp`, which converts a `BPOp` into an `Op` that you can bind with `~$`.

Note that this library doesn't support truly pattern matching on GADTs, and that we had to pass in `Sing bs` as a reference to the structure of our networks.

Gradient Descent

Now we can do simple gradient descent. Defining an error function:

```
errOp
    :: KnownNat m
    => R m
    -> BVar s rs (R m)
    -> BPOp s rs Double
errOp targ r = do
    err <- bindVar $ r - t
    dot ~$ (err :< err :< Ø)
  where
    t = constVar targ
```

And now, we can use `backprop` to generate the gradient, and shift the `Network`! Things are made a bit cleaner from the fact that `Network a bs c` has a `Num` instance, so we can use `(-)` and `(*)` etc.

```
train
    :: (KnownNat a, SingI bs, KnownNat c)
    => Double
    -> R a
    -> R c
    -> Network a bs c
    -> Network a bs c
train r x t n = case backprop (errOp t =<< netOp sing) (x ::< n ::< Ø) of
     (_, _ :< I g :< Ø) -> n - (realToFrac r * g)
```

(`(::<)` is cons and `Ø` is nil for tuples.)

Main

`main`, which will train on sample data sets, is still in progress! Right now it just generates a random network using the *mwc-random* library and prints each internal layer.

```
main :: IO ()
main = withSystemRandom $ \g -> do
```

```haskell
    n <- uniform @(Network 4 '[3,2] 1) g
    void $ traverseNetwork sing (\l -> l <$ print l) n
```

Appendix: Boilerplate

And now for some typeclass instances and boilerplates unrelated to the *backprop* library that makes our custom types easier to use.

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

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

instance (KnownNat n, KnownNat m) => Variate (Layer n m) where
    uniform g = subtract 1 . (* 2) <$> (Layer <$> uniform g <*> uniform g)
    uniformR (l, h) g = (\x -> x * (h - l) + l) <$> uniform g

instance (KnownNat m, KnownNat n) => Num (Layer n m) 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)
    negate (Layer w b) = Layer (negate w) (negate b)
    fromInteger x = Layer (fromInteger x) (fromInteger x)

instance (KnownNat m, KnownNat n) => Fractional (Layer n m) 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 a, SingI bs, KnownNat c) => Variate (Network a bs c) where
    uniform g = genNet sing (uniform g)
    uniformR (l, h) g = (\x -> x * (h - l) + l) <$> uniform g

genNet
    :: forall f a bs c. (Applicative f, KnownNat a, KnownNat c)
    => Sing bs
    -> (forall d e. (KnownNat d, KnownNat e) => f (Layer d e))
    -> f (Network a bs c)
genNet sbs f = go sbs
  where
    go :: forall d es. KnownNat d => Sing es -> f (Network d es c)
    go = \case
      SNil               -> NØ <$> f
      SNat `SCons` ses -> (:&) <$> f <*> go ses

mapNetwork0
    :: forall a bs c. (KnownNat a, KnownNat c)
```

```haskell
    => Sing bs
    -> (forall d e. (KnownNat d, KnownNat e) => Layer d e)
    -> Network a bs c
mapNetwork0 sbs f = getI $ genNet sbs (I f)

traverseNetwork
    :: forall a bs c f. (KnownNat a, KnownNat c, Applicative f)
    => Sing bs
    -> (forall d e. (KnownNat d, KnownNat e) => Layer d e -> f (Layer d e))
    -> Network a bs c
    -> f (Network a bs c)
traverseNetwork sbs f = go sbs
  where
    go :: forall d es. KnownNat d => Sing es -> Network d es c -> f (Network d es c)
    go = \case
      SNil -> \case
        NØ x -> NØ <$> f x
      SNat `SCons` ses -> \case
        x :& xs -> (:&) <$> f x <*> go ses xs

mapNetwork1
    :: forall a bs c. (KnownNat a, KnownNat c)
    => Sing bs
    -> (forall d e. (KnownNat d, KnownNat e) => Layer d e -> Layer d e)
    -> Network a bs c
    -> Network a bs c
mapNetwork1 sbs f = getI . traverseNetwork sbs (I . f)

mapNetwork2
    :: forall a bs c. (KnownNat a, KnownNat c)
    => Sing bs
    -> (forall d e. (KnownNat d, KnownNat e) => Layer d e -> Layer d e -> Layer d e)
    -> Network a bs c
    -> Network a bs c
    -> Network a bs c
mapNetwork2 sbs f = go sbs
  where
    go :: forall d es. KnownNat d => Sing es -> Network d es c -> Network d es c -> Networ
    go = \case
      SNil -> \case
        NØ x -> \case
          NØ y -> NØ (f x y)
      SNat `SCons` ses -> \case
        x :& xs -> \case
          y :& ys -> f x y :& go ses xs ys

instance (KnownNat a, SingI bs, KnownNat c) => Num (Network a bs c) where
    (+)           = mapNetwork2 sing (+)
    (-)           = mapNetwork2 sing (-)
    (*)           = mapNetwork2 sing (*)
    negate        = mapNetwork1 sing negate
    abs           = mapNetwork1 sing abs
    signum        = mapNetwork1 sing signum
```

```haskell
    fromInteger x = mapNetwork0 sing (fromInteger x)

instance (KnownNat a, SingI bs, KnownNat c) => Fractional (Network a bs c) where
    (/)            = mapNetwork2 sing (/)
    recip          = mapNetwork1 sing recip
    fromRational x = mapNetwork0 sing (fromRational x)
```