funn: functional neural networks in haskell

(everything's a category)

Neil Shepperd

August 27, 2015

what is this?

A library for "doing" neural networks. Creating, training, applying.

The goal here is to be hopefully compositional and yet reasonably fast.

`https://github.com/nshepperd/funn`

the problem

A neural network should...

the problem

A neural network should...

1. Take some input $\mathbf{x}$

the problem

A neural network should...

1. Take some input $\mathbf{x}$
2. Take some parameters $\mathbf{W}$

the problem

A neural network should...

1. Take some input $\mathbf{x}$
2. Take some parameters $\mathbf{W}$
3. Produce an output $(\mathbf{y}, \mathcal{L})$

the problem

A neural network should...

1. Take some input $\mathbf{x}$
2. Take some parameters $\mathbf{W}$
3. Produce an output $(\mathbf{y}, \mathcal{L})$
4. Produce a gradient $\frac{d\mathcal{L}}{d(\mathbf{W},\mathbf{x})}$

the solution?

Didn't I just describe the space of differentiable functions?
There's an ekmett library for that™

the solution?

Didn't I just describe the space of differentiable functions?
There's an ekmett library for that™— **ad**.

```
> diff sin 0
1.0
```

Great, we can all go home.

Except...

the solution?

Didn't I just describe the space of differentiable functions?
There's an ekmett library for that™— **ad**.

```
> diff sin 0
1.0
```

Great, we can all go home.

Except...

The performance on thousands of variables is not so great. And we want to be able to export to GPU.

differentiable functions

Let's backtrack a bit.

Differentiable functions *do* form a category.

$$\frac{d}{dx}\mathrm{id}(x) = 1$$

$$\frac{d}{dx}(g \circ f)(x) = f'(x)g'(f(x))$$

(A subcategory of "functions between vector spaces".)

a category

So let's build a category interface.

```haskell
data Network a b = ...

id :: Network a a
(>>>) :: Network a b -> Network b c -> Network a c
```

networks

I spent quite a while experimenting with different designs.

Still not finished yet. But right now:

```haskell
newtype Parameters = Parameters (S.Vector Double)
class Derivable s where
    type family D s :: *
data Network m a b = Network {
  evaluate :: Parameters -> a -> m (b, Double,
                    D b -> m (D a, [D Parameters])),
  params :: Int,
  initialise :: RVar Parameters
}
```

Monad, allowing effects – eg. randomness for dropout units.

Take parameters, produce output together with contribution to loss function and a callback to calculate gradient on the way back.

category interface

```haskell
data Network m a b = ...

id :: Network m a a
(>>>) :: Network m a b -> Network m b c -> Network m a c
```

category interface – id

```haskell
id :: (Monad m) => Network m a a
id = Network ev 0 (return mempty)
 where
  ev _ a = return (a, 0, backward)
  backward b = return (b, [])
```

category interface – (>>>)

```haskell
(>>>) :: (Monad m) => Network m a b -> Network m b c -> Network
(>>>) one two = Network ev (params one + params two) ...
    where ev (Parameters par) !a =
      do let par1 = Parameters (V.take (params one) par)
             par2 = Parameters (V.drop (params one) par)
         (!b, !cost1, !k1) <- evaluate one par1 a
         (!c, !cost2, !k2) <- evaluate two par2 b
         let backward !dc = do (!db, dpar2) <- k2 dc
                               (!da, dpar1) <- k1 db
                               return (da, dpar1 <> dpar2)
         return (c, cost1 + cost2, backward)
```

monoidal category

We're really a sort of monoidal category:

```
left :: Network m a b -> Network m (a,c) (b,c)
right :: Network m a b -> Network m (c,a) (c,b)
(***) :: Network m a b -> Network m c d ->
          Network m (a,c) (b,d)
assocL :: Network m (a,(b,c)) ((a,b),c)
assocR :: Network m ((a,b),c) (a,(b,c))
swap :: Network m (a,b) (b,a)
```

data – statically checked dimensions

Usual unit of data storage for plain neural networks is a fixed length vector.

```haskell
import GHC.TypeLits
import qualified Data.Vector.Storable as S
data Blob (n :: Nat) = Blob { getBlob :: S.Vector Double }
instance Derivable (Blob n) where
    type D (Blob n) = Blob n
```

Using type-level nats we ensure correct construction of the network.

And - dimensions for each layer can sometimes be inferred

data – basic operations

```haskell
fcLayer :: Network m (Blob n1) (Blob n2)
sigmoidLayer :: Network m (Blob n) (Blob n)
quadraticCost :: Network m (Blob n, Blob n) ()
crossEntropyCost :: Network m (Blob n, Blob n) ()
softmaxCost :: Network m (Blob n, Int) ()
```

Softmax is not entirely safe since the domain of the Int is not constrained...

data – sequence combinators

Recurrent neural network: “lifts” a network to act on sequences

```
rnn :: Network m (s,i) (s,o) ->
       Network m (s, Vector i) (s, Vector o)
```

data – sequence combinators

Recurrent neural network: "lifts" a network to act on sequences

```
rnn :: Network m (s,i) (s,o) ->
       Network m (s, Vector i) (s, Vector o)

mapNetwork :: Network m a b ->
              Network m (Vector a) (Vector b)

zipNetwork :: Network m (a,b) c ->
              Network m (Vector a, Vector b) (Vector c)
```

training

Training is by stochastic gradient descent:

```
sgd' :: LearningRate -> Parameters ->
        Network Identity p () ->
        IO p -> IO [Parameters]
```

(IO is just for random selection of training example, should really use a lazy RandT monad instead...)

things i wish i had

Some sort of monadic interface

```
do y <- applyNetwork fcLayer x
   applyNetwork (mergeLayer >>> sigmoidLayer) (x,y)
```