Haskell Net

Matthew Kennedy

January 15, 2020

1 Abstract

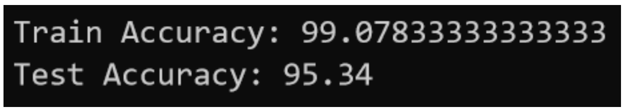
Machine learning code is notoriously hard to test and debug. There is no simple way to check for correctness and there are many places where bugs can occur from incorrect matrix multiplications. The use of dependant types to ensure that the matrix multiplications are valid does not completely solve the issue, but it does go a long way in allowing the compiler help provide a sanity check.

This project is composed of three parts. The first, as discussed, is a library written using dependant types in Haskell to create artificial neural networks. Along with dependant types, the use of type classes to allow for a flexible interface with the library for a user is also discussed. An example network was trained using the MNIST dataset [1] of hand-written numbers. Second, an API was created using Haskell that will accept data to run through the example network. Finally, a front-end was written using Elm, allowing the user to write a number to send to the API. The benefits of using the purely functional framework Elm over other popular options (React) will be evaluated.

2 Summary

The aim of the project was to create a machine learning library that takes advantage of dependent types to create type-safe neural networks and to create a web front-end and API to interact with the library.

A successful outcome for the project was defined as being able to train a network using the MNIST dataset, achieving a prediction accuracy of greater than 95% and then build the aforementioned front-end and API.



```
Train Accuracy: 99.07833333333333
Test Accuracy: 95.34
```

Figure 1: Results from training a neural network on the MNIST dataset

These goals have been met. The library was used to train a network using MNIST to greater than 95%. An API was written using the Servant and Aeson libraries which takes JSON data containing an array for the data of the number to run through the trained network. Elm along with some JavaScript was used to write a front-end that sends a number written by the user to the API. This front-end is deployed here.

3 Background on Neural Networks

Before discussing the implementation it is important to understand roughly how a neural network works.

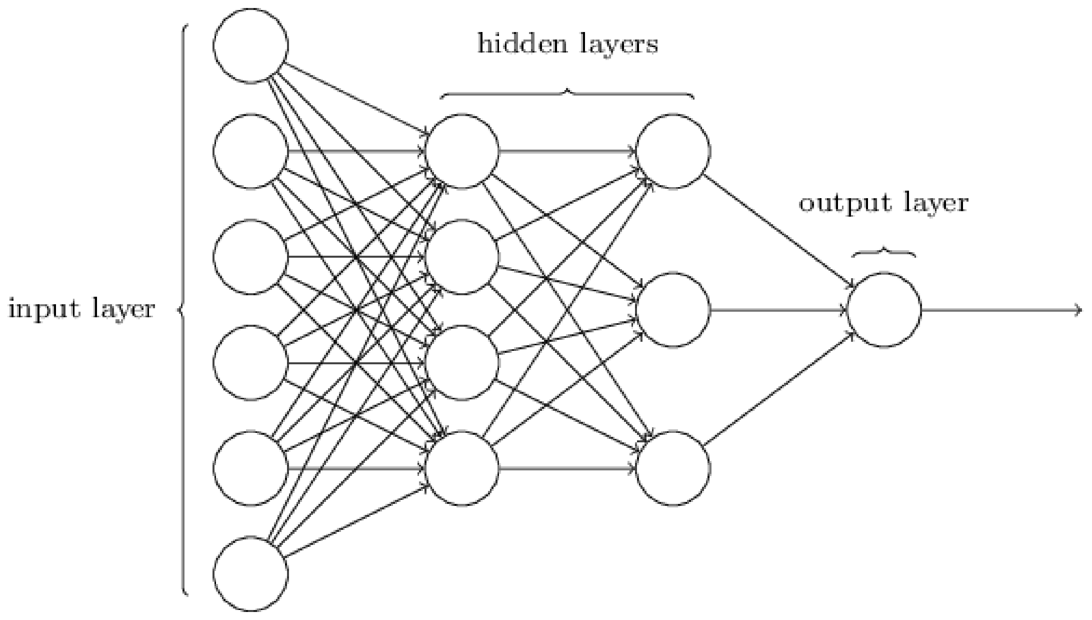


Figure 2: Diagram of artificial neural network [2]

The circles in figure 2 will be referred to as perceptrons. These perceptrons are organised in layers. The networks are fully-connected, therefore any given perception is connected to every perceptron in adjacent layers. These connections are represented by the arrows in figure 2.

The important state within the network is on the connections, this state is commonly referred to as a weight. The perceptrons do not actually have any lasting state, the value is dependant on the data used in the input layer and the values for the weights of the connections. It is worth noting that in practice there is an extra connection for each perceptron other than those in the input layer, this is referred to as bias.

There are two main steps to understand in neural networks: forward and backwards propagation. Forward propagation is simply the process of running data through the network to obtain a result. Backwards propagation takes the result obtained from forward propagation and compares it to the expected result. This comparison is often called the error or cost. It uses this error and goes in reverse through the network, calculating gradients and using them to update the weights of the connections. The next time data is run through the network the error should be less meaning the result is closer to the expected result, the performance of the network has been improved.

I will not go into the details on the maths behind forwards and backwards propagation, there are many good resources online if interested [2]. However, there are a few parts of the network I want to highlight as not only are they important for the performance of the network, but I'm also going to discuss how they are implemented in the library.

3.1 Initialisation

When you construct a network the weights need to have some initial state, this is referred to as initialisation. It turns out this initialisation is very important for the performance of neural networks [3]. I'm not going to go into the details, if interested see [4].

3.2 Activation Functions

Activation functions are a key part of neural networks. They add a non-linear property to a neural network, allowing it to model more complex relationships and patterns in the data. The activation function is applied after every set of connections in forward propagation, for more details see [5]. The gradient of the activation function is also used during backwards propagation and can affect the choice of cost function (the cost function is used to understand how far the results obtained are from the expected results).

3.3 Optimisation Function

Optimisation functions are used during backwards propagation. It is the function that takes the gradients calculated and uses them to update the weights of the connections. The most simple optimisation function is called gradient descent. In this function, you simply calculate the gradients and subtract them from the current weights. While this method has proved to be powerful, there are better ones. The first of these is stochastic gradient descent. This method is basically the same as standard gradient descent except the gradients are accumulated over multiple inputs and the average is subtracted to update the weights. The optimisation function is key for speed and performance of learning. More information can be found at [6].

4 HaskellNet

4.1 Introduction

This project was largely inspired by a blog series on type-safe neural networks by Justin Le [7]. The code for HaskellNet used the series as a foundation to work from.

Haskell is not known for high performance linear algebra and using standard Haskell to implement the networks would result in slow performance. Fortunately, there is a library called HMatrix [8] that uses OpenBLAS to allow for efficient numeric computation. This library also has a static extension which provides dependently type vectors and matrices and in fact does a lot of the heavy lifting for the neural network implementation. If interested in how dependently typed vectors can be implemented there is a good blog at [9].

HMatrix is used throughout the library and it is therefore important to know about a couple of data types. The first of these is *R x*, this type represents a vector of size x. The second is *L x y*, this represents a matrix with x rows and y columns. It also provides all of functions to operate on the vectors/matrices. These will be noted when seen in the code.

HaskellNet is split up into a few modules:

- Activations.hs
- Connections.hs
- Init.hs
- Network.hs
- Optimiser.hs
- Params.hs
- Train.hs

4.2 Activations.hs

This module exposes a typeclass *Activations* that has functions every activation function needs to implement. They are *calcActivations*, *calcGradients* and *calcCost* and will be used within the Network.hs module.

```
class Activation a where
  calcActivations :: (KnownNat b) => a -> R b -> R b
  calcGradients :: (KnownNat b) => a -> R b -> R b
  calcCost :: (KnownNat b) => a -> R b -> R b -> Double
```

It also has a few predefined activation functions. These are represented as data types that are made into instances of the activations type class. One example of this can be seen with the Sigmoid activation function.

```
data Sigmoid =
  Sigmoid

instance Activation Sigmoid where
  calcActivations Sigmoid x = 1 / (1 + exp (-x))
  calcGradients Sigmoid x =
    let x' = calcActivations Sigmoid x
    in x' * (1 - x')
  calcCost Sigmoid target a =
    sumElements' $
      (target * log a) + ((1 - target) * log (1 - a))
```

4.3 Connections.hs

Connections.hs exposes the data type used to represent the connections in a neural network. In this case the *i* refers to the number of perceptrons in the input layer to the connections and *o* to the number in the output layer. Along with the *weights* and *biases* fields expected, there are also the *bGrads* and *wGrads* fields. These are used to store the biases gradients and weights gradients generated during back propagation.

```
data Connections i o =
  C
  { biases  :: !(R o)
  , weights :: !(L o i)
  , bGrads  :: !(R o)
  , wGrads  :: !(L o i)
  }
  deriving (Show, Generic)
```

The module also contains functions related to Connections. The first will take a biases and weights gradient then add it to the current value.

```
updateGrads ::
  (KnownNat i, KnownNat o)
  => R o
  -> L o i
  -> Connections i o
  -> Connections i o
updateGrads bGrads' wGrads' (C b w bGrads wGrads) =
  C b w (bGrads + bGrads') (wGrads + wGrads')
```

The second is used when running a forward pass on the network.

```
weightedInput ::
  (KnownNat i, KnownNat o)
  => Connections i o
  -> R i
  -> R o
weightedInput (C b w _ _) v = w #> v + b
```

The *weightedInput* function is used during forward propagation. It multiplies some input data by the weights matrix and adds on the bias.

The *(#>)* operator is a dense matrix-vector product. It is defined in HMatrix's Static library as follows.

```
(#>) :: (KnownNat m, KnownNat n) => L m n -> R n -> R m
```

If the data passed to the function is not of the correct size the compiler will throw an error. This is the same for all the operators defined in the HMatrix Static library. It may seem like it is a bit overkill, however, it helps to eliminate a class of bugs that can be very hard to catch and can act as a sanity check for your code. There were a few times when writing my code that I made a mistake, instead of having the pain of getting weird outputs from the network and trying to work out exactly where they came from, the compiler flagged it immediately and I could solve the bug easily.

4.4 Init.hs

Similar to the Activations.hs module, Init.hs contains a type class *Init* that has a function that will initialise a set of connections, as discussed in section 3.1.

```
class Init a where
  genRandom ::
    (KnownNat i, KnownNat o)
    => a
    -> IO (Connections i o)
```

It also has a couple of predefined initialisation functions. One of which being Kaiming initialisation [10].

```
data Kaiming =
  Kaiming

instance Init Kaiming where
  genRandom Kaiming = kaimingInit
```

```
kaimingInit ::
  (KnownNat i, KnownNat o)
  => IO (Connections i o)
kaimingInit = do
  let b = 0 * randomVector 0 Uniform
  w <- randn
  let w' = sqrt (2 / fromIntegral (snd $ size w)) * w
  return (C b w' 0 0)
```

Note that the size of the vectors is never explicitly used in the code. Haskell can just infer the size required based on the type.

4.5 Params.hs

Simply provides a data type that contains all of the hyper parameters used when training a network. These are batch size and learning rate. Batch size is the number of inputs that is run through the network before applying the optimisation function. Learning rate scales the amount gradients affect the current values on the connections. A higher learning rate means faster training, but go too high and the network will start to diverge away from the optimum.

```
data Params =
  Params
  { bs :: !Int
  , lr :: !Double
  }
```

4.6 Optimiser.hs

This is another module that exposes a type class. It is called *Optimiser* and has a function that will run the optimiser over a network, updating the weights and biases of the connections to improve the performance of the network.

```
class Optimiser a where
  runOptimiser ::
  (KnownNat i, KnownNat o)
  => a
  -> Params
  -> Connections i o
  -> Connections i o
```

There are a couple of instances already defined: *GradientDescent* and *Sgd* (Stochastic Gradient Descent) as were discussed in section 3.3. The *Sgd* instance of the *Optimiser* type class is seen below.

```
data Sgd =
  Sgd

instance Optimiser Sgd where
  runOptimiser Sgd = sgd

sgd ::
  (KnownNat i, KnownNat o)
  => Params
  -> Connections i o
  -> Connections i o
sgd (Params bs lr) (C b w bGrads wGrads) =
  C (b - (konst (lr / fromIntegral bs) * bGrads))
  (w - (konst (lr / fromIntegral bs) * wGrads))
  0
  0
```

4.7 Network.hs

This module has the data type that represents a neural network and all of the related functions. It uses the Activations, Connections, Init, Optimiser and Params modules as dependencies and contains most of the logic behind the library.

4.7.1 Data Type

To start with let us look at the data type for a network, it is defined using GADT syntax.

```
data Network :: Nat -> [Nat] -> Nat -> * where
  Output :: !(Connections i o) -> Network i '[] o
  Layer
    :: KnownNat h
    => !(Connections i h)
    -> !(Network h hs o)
    -> Network i (h : hs) o
```

There are two constructors: *Output* and *Layer*. *Output*, of course, refers to the output layer of a neural network. As it is the final layer it will produce a network with no hidden layers, simply a single set of connections. Layer can be considered as 'consing' extra layers on the front of the output layer. It will take a new set of connections and add them to the front of an existing network. Let us take the network used to to achieve 95% on the MNIST dataset [1]. The dataset consists of 28x28 pixel images so the input layer will have 784 perceptrons, there is only one hidden layer with 100 perceptrons and an output layer that has 10 perceptrons (1 for each number 0-9). This would be represented as having type *Network 784 '[100] 10*, which is equivalent to *Connections*

784 100 ‘Layer‘ Output (Connections 100 10).

4.7.2 Initialisation

The next function is used to initialise a network. To understand it some background knowledge is required on singletons. I am not going to go into too much detail, just explain the rationale behind using singletons at a high level. One big problem with trying to use dependent types in Haskell is that the types are erased at runtime, meaning we cannot directly use the type as a value. This is a problem because we cannot then pattern match on the type to do things. This is where singletons come in, they provide a work around for this issue. I consider it to be a bit of a hack honestly and not extremely pleasant to work with, it would be much nicer if Haskell didn't put up so much resistance but it is the best that can be done currently.

The *initialiseNet* function uses singletons to pattern match on the hidden layers of the network. It can be thought of as being as exactly the same as you would normally pattern match on lists. *SNil* being equivalent to *[]* and *SNat ‘SCons‘ ss* to *x:xs*.

Once this is understood it is not too hard to understand, at least conceptually what the function is doing. If the hidden layers list is empty we *fmap* the *Output* constructor onto a set of initialised connections (*fmap* is needed as *genRandom* will return connections in *IO*). If the list is not empty the connections are passed to the layer constructor along with a recursive call that will produce the rest of the network.

Also note how the type class *Init* is being used, anything can be passed as an argument to the initialise net function as long as it is an instance of the *Init* class.

```haskell
initialiseNet ::
  (KnownNat i, SingI hs, KnownNat o, Init f)
  => f
  -> IO (Network i hs o)
initialiseNet = go sing
  where
    go ::
      (KnownNat h, KnownNat o, Init f)
      => Sing hs'
      -> f
      -> IO (Network h hs' o)
    go sing init =
      case sing of
        SNil            -> Output <$> genRandom init
        SNat `SCons` ss ->
          Layer <$> genRandom init <*> go ss init
```

4.7.3 Inference

The *runNet* function runs some input data through a forward pass of a network and returns the outputs. This is the function that will be used to produce predictions once a network has been trained, known as inference.

Again note the uses of the *Activation* type class to allow flexibility as to what is being used for the activation functions in the network.

```
runNet ::
  (KnownNat i, KnownNat o, Activation outA, Activation intA)
  => Network i hs o
  -> ModelActivations outA intA
  -> R i
  -> R o
runNet (Output c) (ModelActivations outAct _) !inp =
  calcActivations outAct (weightedInput c inp)
runNet (c `Layer` n) ma@(ModelActivations _ intAct) !inp =
  let inp' = calcActivations intAct (weightedInput c inp)
  in runNet n ma inp'
```

4.7.4 Helper Data Types

The data is simply represented in a data type *DataBunch* that takes two lists of vectors. One is the inputs to the network and the other is the labels (expected results) from the network. Dependant types will help us out here, the size of the input and label vectors will have to match the corresponding size of the network created, if not it won't compile.

There is a second data type *ModelData* that has two *DataBunch* types. The first is the data that is actually going to be used to train the network and the second is data that is going to be used to check that the model is not overfitting to the training data.

```
data DataBunch i o =
  DataBunch
  { inputs :: ![R i]
  , labels :: ![R o]
  }
  deriving (Show)

data ModelData i o =
  ModelData
  { trainingData :: !(DataBunch i o)
  , testData     :: !(DataBunch i o)
  }
  deriving (Show)
```

Often, it is desired to have a different activation function on the output layer than the internal layers. The *ModelActivations* data type allows for this.

```
data ModelActivations outA intA =
  ModelActivations !outA !intA
```

4.7.5 Training

The training code is fairly long but not much of it interesting from a functional programming perspective, as a result I will not go into too much depth, just give a high level understanding of what each section is doing. In most cases the type signatures of the functions have also been omitted to make the code more concise.

At the highest level there is the *runEpochs* function. This function is what is called by the user to train a network. It will take in all the data needed and will run training for a specified number of epochs. One epoch is simply running all of the training data through a network once. For example, with the MNIST dataset there are 50000 training inputs, one epoch is complete when all 50000 have been run through the network.

```
runEpochs numEpochs md ma ps optType net =
  foldl' (epoch md ma ps optType) net [1 .. numEpochs]
```

The code to run an individual epoch will call the training function and print out some useful information for the user. Namely the cost (value that specifies how far obtained results are from expected results), training accuracy and test accuracy, which represent the percentage of inputs the network is producing correct predictions for on the training and test datasets respectively.

```
epoch (ModelData trainD testD) ma ps optF netIO count = do
  net <- netIO
  putStrLn $ "Running epoch " ++ show count ++ "..."
  let (net', cost) = train trainD ma ps optF net
      trainAcc = getAccuracy trainD 0.8 ma net'
      testAcc = getAccuracy testD 0.8 ma net'
  putStrLn $ "Cost: " ++ show cost
  putStrLn $ "Train Accuracy: " ++ show trainAcc
  putStrLn $ "Test Accuracy: " ++ show testAcc
  putStrLn ""
  return net'
```

The *getAccuracy* function runs all of the inputs through the network and compares the obtained results to the target results. It will then return a percentage for the number the network gets correct.

```
getAccuracy (DataBunch inps targets) thresh ma net =
  let numCorrect =
        foldl'
          (\n (inp, target) ->
            if LA.cmap
                (\x ->
                  if x >= thresh
                   then 1
                   else 0)
                (unwrap $ runNet net ma inp) ==
              unwrap target
             then n + 1
             else n)
        0
        (zip inps targets)
  in (numCorrect / fromIntegral (length inps)) * 100
```

The *train* function calls an auxiliary function *handleBatching*. The handle batching will accumulate the cost over a number of inputs so the average is calculated and returned, along with the updated network.

```
train db@(DataBunch inps _) ma ps optType net =
  let (_, net', costSum) =
        handleBatching db ma ps optType net 0
  in (net', -(costSum / (2 * fromIntegral (length inps))))
```

All *handleBatching* does is run multiple inputs through the network before optimising. It does this by splitting the training data into a smaller 'batch' and passing that to *backPropagate*. This happens recursively until all training data has been used.

```
handleBatching (DataBunch [] []) _ _ _ net cost =
  (DataBunch [] [], net, cost)
handleBatching (DataBunch inps targets) ma ps@(Params bs lr)
    optType net cost =
  let (batchInps, inps') = splitAt bs inps
      (batchTargets, targets') = splitAt bs targets
      (net', cost') =
        backPropagate
          (DataBunch batchInps batchTargets)
          ma
          net
      optimisedNet = optimise ps optType net'
  in handleBatching (DataBunch inps' targets') ma ps optType
                    optimisedNet (cost + cost')
```

The *backPropagate* function takes the small 'batch' of inputs and uses a fold, running each input individually using *backPropOneInput*.

```
backPropagate db@(DataBunch inps targets) ma net =
  let (netFinal, costSum) =
    foldl'
      (\(net, cost) (inp, target) ->
        let (net', cost') = backPropOneInput inp target ma
 net
        in (net', cost + cost'))
      (net, 0)
      (zip inps targets)
  in (netFinal, costSum)
```

As mentioned, after back propagating over a batch a network is optimised as discussed in section 3.3. This is done using the *optimise* function. Note the type signature has been included to show use of the *Optimiser* type class. All the function does is recursively move through the network, applying the optimisation function to each set of connections.

```
optimise ::
  (KnownNat i, KnownNat o, Optimiser opt)
  => Params
  -> opt
  -> Network i hs o
  -> Network i hs o
optimise ps optType (Output c)   =
  Output (runOptimiser optType ps c)
optimise ps optType (c `Layer` n) =
  runOptimiser optType ps c `Layer` optimise ps optType n
```

The *backPropOneInput* and auxiliary functions is where most of the linear algebra behind the neural network occurs. The function itself does very little, just makes a call to the auxiliary function *go*.

```
backPropOneInput !inp !target ma net =
  let (net', _, cost) = go inp target ma net
  in (net', cost)
```

The *go* function also has the type signature included, this time to highlight the use of the *Activations* type class. Note the *internalErrs* function, which will be seen shortly, also makes use of the type class.

This function pattern matches on the network. If on the output layer the input data is run through the connections and the activations calculated. This is then used to calculate the output errors which is effectively the starting state for back propagation. The cost is also calculated here, note the cost does not actually have any effect on training, it is simply a feedback mechanism for the user.

If on an internal layer the calculations are largely the same except the internal activation function is used and the internal errors are calculated.

```
go ::
  (KnownNat i, KnownNat o, Activation outA, Activation intA)
  => R i
  -> R o
  -> ModelActivations outA intA
  -> Network i hs o
  -> (Network i hs o, R i, Double)
go !inp !target (ModelActivations outAct _) (Output c) =
  let z = weightedInput c inp
      a = calcActivations outAct z
      (c', dWs) = outputErrs inp target a z c
      cost = calcCost outAct target a
  in (Output c', dWs, cost)
go !inp !target ma@(ModelActivations _ intAct) (c `Layer` n)
    =
  let z = weightedInput c inp
      a = calcActivations intAct z
      (c', n', cost, dWs) = internalErrs inp target c ma n a
    z
  in (c' `Layer` n', dWs, cost)
```

The *outputErrs* function calculates the error for the output layer and updates the gradients accordingly. It also calculates *dWs*, which is returned as it will be used by the previous layer in the network to calculate its errors.

```
outputErrs !inp !target !a !z c@(C _ w _ _) =
  let errs = (a - target)
      errWs = errs `outer` inp
      dWs = tr w #> errs
  in (updateGrads errs errWs c, dWs)
```

The *internalErrs* function recursively calls the *go* auxiliary function. This is because back propagation effectively has to start at the output layer. So the *go* function is called till the output layer is reached. The data returned from the output layer then allows the layers to resolve in reverse. Again, the gradients are updated accordingly for each layer and the *dWs* returned.

```
internalErrs !inp !target c@(C _ w _ _) ma@(ModelActivations
     _ intAct) net !a !z =
  let (net', dWs', cost) = go a target ma net
      errs = dWs' * calcGradients intAct z
      errWs = errs `outer` inp
      dWs = tr w #> errs
  in (updateGrads errs errWs c, net', cost, dWs)
```

4.7.6 Serialisation

Training a network can take a long time and need a lot of data, it would not be good to have to retrain it every time we wanted to use it. Therefore, it is very important to have some ability to save a trained network. This can be done in Haskell by making our *Connections* and *Network* data types into instances of the *Binary* type class. This code is exactly as in Justin Le's blog post [11] so I take no credit.
The *Connections* data type derives *Generic* from *GHC.Generics*. This allows for it to easily be made into an instance of the *Binary* type class.

```
instance (KnownNat i, KnownNat o) => Binary (Connections i o
    )
```

As the *Network* data type is a GADT it has to be done manually. Fortunately, it is straight forward.

For *put* it can be done by pattern matching on the constructor and taking advantage of the fact that the connections are already an instance of *Binary*.

```
putNet :: (KnownNat i, KnownNat o) => Network i hs o -> Put
putNet =
  \case
    Output w -> put w
    w `Layer` n -> put w *> putNet n
```

For *get* we again pattern match but this time have to use singletons for reason discussed in section 4.7.2.

```
getNet :: (KnownNat i, KnownNat o) => Sing hs -> Get (
   Network i hs o)
getNet =
  \case
    SNil -> Output <$> B.get
    SNat `SCons` ss -> Layer <$> B.get <*> getNet ss
```

Then making the *Binary* instance is simple.

```
instance (KnownNat i, SingI hs, KnownNat o) => Binary (
   Network i hs o) where
put = putNet
get = getNet sing
```

4.8 Train.hs

This module does not actually contain any code that effects the behaviour of the library. It instead contains a couple of functions that are examples of actually training a network. If wanting to see how the library comes together from the perspective of a user it may be interesting to have a look through the source code.

4.9 Type Classes

When developing and experimenting with the library it became apparent that it was desirable that certain parts of the code were flexible, so that the behaviour could be changed depending on the situation. There are a few different ways of achieving this, the best of which I found to be type classes. This section discusses the different possible approaches and why type classes were decided upon as the best option.

4.9.1 Hard Coding

One way of changing the behaviour is to go into the code and edit it to include the functions wanted. This requires a full understanding of the code and results in having to change the same thing in multiple places. This approach is obviously not friendly to a user of the library.

4.9.2 Functions as Arguments

This approach is better than the first. The code can take functions as arguments instead of having them hard coded in. This allows the desired functions to be decided upon when using the library and also has the benefit of allowing the user to get creative with the functions they use. If the function has the desired behaviour it can be used. However, there is a problem with this approach. It can result in having to have many more arguments to the function than desired. For example, imagine every time you wanted to add to numbers you had to pass in the add function as an argument, not very friendly behaviour.

4.9.3 Pattern Matching on a Type

Effectively, we want to only have one function that can have multiple behaviours depending on the type. Pattern matching on a type can allow us to do this. Let's take the activation functions as an example.

```
data Activation =
  Sigmoid
  | Relu
  | Softmax

calcActivations :: (KnownNat a) => Activation -> R a -> R a
calcActivations activation x =
  case activation of
    Sigmoid -> ...
    Relu -> ...
    Softmax -> ...

calcGradients ...
calcCost ...
```

Now in the network code only the type of activation that is wanted needs to be passed. One type has to be provided instead of three separate functions. This approach can work great if all possible behaviours are known or it is not desirable to provide flexibility, but not so great if the user should be able to add their own custom activation functions. They would have to dive into the source code and add new constructors to the type and new cases to the functions.

4.9.4 Type Classes

Type classes will have all the properties that pattern matching has with the added benefit of providing flexibility. The user has no need to edit the library code, they can make there own activation functions outside of the library and use them in a network, exactly the desired behaviour.

4.10 Dependant Types

There are some clear benefits in using dependant types for neural networks. It does eliminate one source of bugs in the programs and in many cases type inference can be used to write the logic for us, without having to worry about the size. The only place it needs to be defined correctly is on the type itself.

Haskell makes dependant types hard by erasing the types at runtime so they cannot be used as values by the program. A way round this is to use singletons but this is not particularly friendly or intuitive.

As a result of this it is probably best to not start off with dependant types in a project. Instead start building and if it becomes clear that dependant types will make a significant difference, add them in later.

5 Server

There isn't too much interesting with the API code. The Servant library makes good use of types [12], but I will not go into details here.

The one thing to show is how simple it is to use a previously trained network. The main function simply de-serialises a saved network and passes it to the handler. Note that the API has been deployed on AWS to make it available to the front-end.

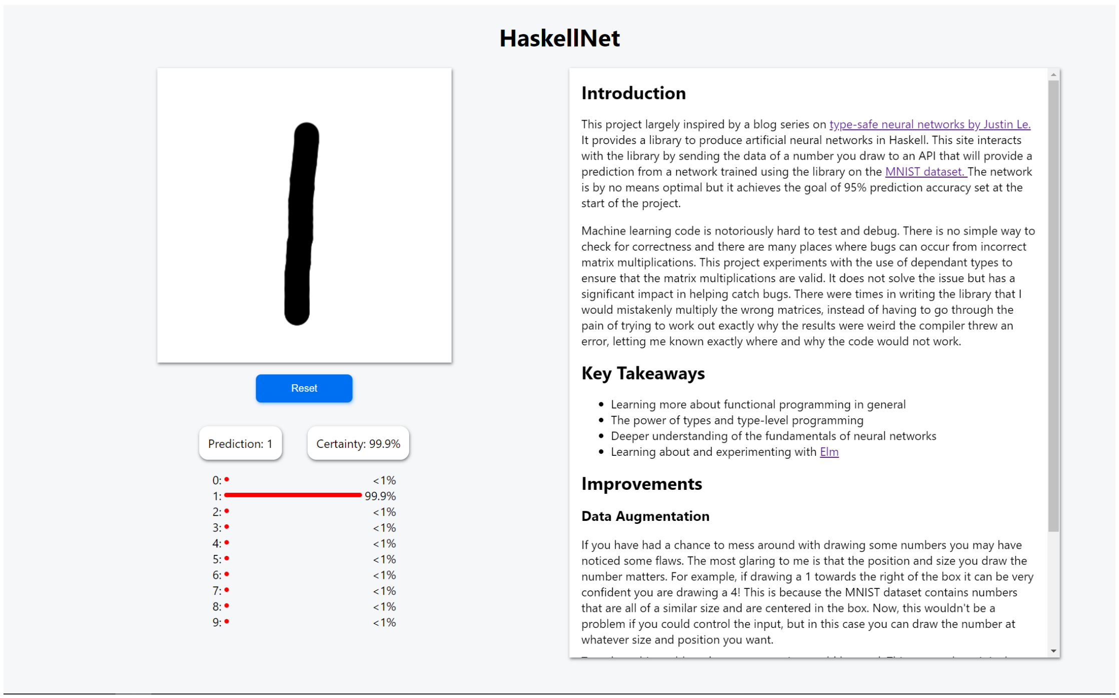
```
main :: IO ()
main = do
  net <- loadNet "./data/mnistNetSigmoid.txt" :: IO (Network
    784 '[ 100] 10)
  run 8081 (application net)
```

When JSON data is received it will end up with *getPrediction* which uses the *runNet* (see section 4.7.3) function to get the predictions from the model. Note the predictions are wrapped in the *Prediction* data type which is just use to convert it back to JSON to send to the client.

```
getPrediction :: Network 784 '[ 100] 10 -> NumberString ->
    Prediction
getPrediction net (NumberString img) =
  Prediction
    ((LA.toList . unwrap) $
      runNet net (ModelActivations Sigmoid Relu) (
    convertToVec img))
```

6 Front-end

A front-end was creating using Elm. It provides a box to write a number that will be sent off to the API and some visualisations to view the results. It also includes some information about the project on the right hand side.



Introduction

This project largely inspired by a blog series on type-safe neural networks by Justin Le. It provides a library to produce artificial neural networks in Haskell. This site interacts with the library by sending the data of a number you draw to an API that will provide a prediction from a network trained using the library on the MNIST dataset. The network is by no means optimal but it achieves the goal of 95% prediction accuracy set at the start of the project.

Machine learning code is notoriously hard to test and debug. There is no simple way to check for correctness and there are many places where bugs can occur from incorrect matrix multiplications. This project experiments with the use of dependant types to ensure that the matrix multiplications are valid. It does not solve the issue but has a significant impact in helping catch bugs. There were times in writing the library that I would mistakenly multiply the wrong matrices, instead of having to go through the pain of trying to work out exactly why the results were weird the compiler threw an error, letting me known exactly where and why the code would not work.

Key Takeaways

- Learning more about functional programming in general
- The power of types and type-level programming
- Deeper understanding of the fundamentals of neural networks
- Learning about and experimenting with Elm

Improvements

Data Augmentation

If you have had a chance to mess around with drawing some numbers you may have noticed some flaws. The most glaring to me is that the position and size you draw the number matters. For example, if drawing a 1 towards the right of the box it can be very confident you are drawing a 4! This is because the MNIST dataset contains numbers that are all of a similar size and are centered in the box. Now, this wouldn't be a problem if you could control the input, but in this case you can draw the number at whatever size and position you want.

Figure 3: Screenshot of the front-end built with Elm

Again, I'll not go into too much detail. I shall discuss how using Elm, which is a purely functional FRP framework, has advantages over popular counter parts, namely Facebook's React. To do this the example of how the canvas to write the number was written.

The drawing of a number on a canvas can be considered unsafe. It requires mutation of the state said canvas. Therefore, it is not possible to code it directly in Elm. What Elm does provide however is an intuitive API for JavaScript interop. A port can be set up that allows data to be sent directly from JavaScript to Elm. When Elm receives the data it can update its state accordingly. This has the effect of isolating the unsafe component from the rest of the application logic, ensuring errors do not cause chaos. In addition, what is being sent from JavaScript can be sanity checked. It will be caught on entry and an error thrown, rather than appearing further down the line, causing some unexpected behaviour. For example in this case, all the application knows is that it may receive an array from JavaScript at some point and the logic to handle the array when it does. It has no knowledge of how the array was produced.

Comparatively, if using React, anything you can do with JavaScript you can do inside the framework. Before learning more about functional programming and experimenting with Elm this is exactly what I would have done. I would have had the unsafe canvas code embedded inside a react component somewhere. In fact, even now I would have to think about exactly how to make the code safe. With Elm there is no thinking to be done, it forces safety.

7 Conclusion

This project has allowed more in depth exploration of functional programming through building a real world application. There was a particular focus on experimenting with types, both in terms of how dependant types and how more simple type classes can be used to improve code. Using Elm also highlighted how purely functional code can save you a lot of pain further down the line by putting up barriers, forcing code to be written in a safe way.

Whilst the project did highlight some of the benefits dependant types can provide, it also showed me that the implementation in Haskell is fairly convoluted. It requires some trickery to pattern match on the types at run-time. Therefore, in its current state I do not plan on using them in future applications unless the benefits are significant.

In addition, the exercise of building a neural network library from scratch has given a deeper understanding of the fundamental concepts and allowed me to achieve the goal of training a network to get 95% accuracy on the MNIST dataset.

The front-end and API have both been deployed. The front-end available on GitHub pages and API on an AWS instance.

References

[1] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[2] M. A. Nielsen, "Neural networks and deep learning," 2018. [Online]. Available: http://neuralnetworksanddeeplearning.com/

[3] D. Mishkin and J. Matas, "All you need is a good init," *arXiv preprint arXiv:1511.06422*, 2015.

[4] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: http://proceedings.mlr.press/v9/glorot10a.html

[5] A. Oppermann, "Activation functions in neural networks," 2019. [Online]. Available: https://www.deeplearning-academy.com/p/ai-wiki-activation-functions

[6] ——, "Optimization algorithms in deep learning," 2019. [Online]. Available: https://www.deeplearning-academy.com/p/ai-wiki-optimization-algorithms

[7] J. Le, "Practical dependent types in haskell: Type-safe neural networks (part 1)," 2016. [Online]. Available: https://blog.jle.im/entry/practical-dependent-types-in-haskell-1.html

[8] *hmatrix: Numeric Linear Algebra*. [Online]. Available: https://hackage.haskell.org/package/hmatrix

[9] H. ISHII, "Dependent types in haskell," 2014. [Online]. Available: https://www.schoolofhaskell.com/user/konn/prove-your-haskell-for-great-safety/dependent-types-in-haskell

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015. [Online]. Available: http://arxiv.org/abs/1502.01852

[11] J. Le, "Practical dependent types in haskell 2: Existential neural networks and types at runtime," 2016. [Online]. Available: PracticalDependentTypesinHaskell2:ExistentialNeuralNetworksandTypesatRuntime

[12] "Why is servant a type-level dsl?" [Online]. Available: https://www.servant.dev/posts/2018-07-12-servant-dsl-typelevel.html