Week 5: The Backpropagation Algorithm

Hans Georg Schaathun

25th April 2017

Time	Topic	Reading
8.15-	Recap of tutorials last week	
	Lecture: From perceptron to back-propagation	Marsland Section 4
9.00-	Tutorial 5.1: The Backpropagation Algorithm	
11.45-	Lunch break	
12.15-	Questions and answers	
13.00-	Tutorial 5.1 continued	

This PDF document is available in an HTML version at `http://www.hg.schaathun.net/FPIA/week05.html`.

1 Tutorial 5.1: The Backpropagation Algorithm

Reading: Stephen Marsland, Chapter 4

Today, you need to implement and test the backpropagation algorithm as described in Marsland's book. You can build on the perceptron from last week, as well as the multi-layer network which you used on the XOR problem in Tutorial 7. However, it is a good idea to take a copy of what you have before making changes.

1.1 Tasks

We up the challenge compared to last week. With spoon feeding out the window, you need to manage the big picture yourselves. The objective is straight-forward, to implement back-propagation to train a multi-layer neural network. Several smaller tasks must be covered, including:

First we focus on training the network using only one item from the training set.

- Update each output weight w_i, using
  - $w_i := w_i - \eta\delta x'_i$
  - where $\delta = (y - t)y(1 - y)$
- Update each hidden layer weight $v_{i,j}$, using
  - $v_{i,j} := v_{i,j} - \eta\delta'_i x_j$
  - where $\delta'_i = x'_i(1 - x'_i)\delta w_i$
- If there are multiple output neurons,
  - w_i is updated for each output neuron independently
  - $\delta'_i = x'_i(1 - x'_i)\sum \delta_o w_{o,i}$
    - where o ranges over the output nodes

Figure 1: The Backpropagation algorithm.

```
trainNetworkOne :: Double -> Network -> (Double,[Double]) -> Network
```

This is a big problem which you need to break up into subproblems.

1.1.1 Task 1: New threshold function

You need to adopt the sigmoid instead of the hard threshold function.

1.1.2 Task 2: Forward pass

The forward pass is essentially the recall. However, you are going to need the output from every layer in the backward pass. Hence you need to store the inputs for each layer as well as the final output. Thus you get an object of type [[Double]], instead of just the [Double] that you would get from the output layer.

You can choose between two options.

1. Assume two layers `[hl,ol]` and calculate the intermediate output/input `xs'` and final output `ys`, and return `[xs,xs',ys]`.
2. Do the forward pass *recursively* for an arbitrary number of layers.

1.1.3 Task 3: Backward pass

The formulæ for backpropagation are shown in Figure 1. Contrary to the previous talk, it includes the case with multiple output nodes.

Observe what variables you need when you update the weights in the hidden layer. You need the input and output for the layer, as well as the δs from the output layer. It follows that you need to calculate and keep the δs and feed them to the previous layer. You also need the intermediat input/output from the forward pass.

It is possible to implement backpropagation for an arbitrary number of layers, but it takes significant experience with recursion. Here we will only indicate a solution for two-layer networks.

Output layer Starting on the backward pass, we have a list of neurons (the layer), and a list of outputs ys (one y per neuron).

1. Start by making a function to update a single neuron, based on the existing neuron and corresponding y. It needs to output a pair, comprising the new neuron and the δ used in the calculation.
2. You can use `zipWith` and the function you just made to update a complete layer. The output is a list of neuron/δ pairs.

Accumulating δs So far we have a list of neuron/δ pairs. This is required to update the weights in the hidden layer, the but only in the factor $\sum \delta w_i$. Each hidden neuron has a different value for w_i, and thus a different sum.

You need a function to calculate these sums, and return a list with one sum for each neuron in the hidden layer.

Hidden layer Starting on the hidden layer, you have four important pieces of information.

1. The input to the network
2. The output from the hidden layer as calculated in the forward pass.
3. The sums of δs calculated above.
4. The hidden layer itself, with neurons and their weights

To update weights, start with a function to do a single neuron; then proceed with the complete layer using map, zipWith, recursion, or some similar technique.

Completing the backpropagation algorithm Armed with the functions designed above, you can write the `trainNetworkOne` function to train the full network:

```
trainNetworkOne :: Double                  -- Learning rate (eta)
                -> Network                 -- Old network
                -> (Double,[Double])       -- One data item (row)
                -> Network                 -- Updated network
```

1.1.4 Task 4: Training on a full training set

```
epoch :: Double -> Network -> [(Double,[Double])] -> Network
epoch eta n samples = foldl' (trainNetworkOne eta) n samples
```

Question. *What does the `foldl'` function do?* Use google or check the textbook.

1.1.5 Task 5: Many iterations

To fully train a network, multiple epochs are required. We need a function to recursing to run epoch n times.

More advanced stop criteria are possible, running a variable number of epochs depending on how quickly the weights converge.

1.1.6 Task 6: Testing

Test the classifier, using the ideas discussed in Tutorial 6.

The following is an example of an executable module to test a classifier. It prints error rates to the standard output. You need to define four functions to use it (see comments).

```
module Main where

import ANNData          == your module to load CSV data
-- it should define mkTestTrainSets
import NeuralNetwork  == your module defining a backpropagation network
-- it should define:
--     initNetwork
--     trainNetwork
--     testNetwork

p = 0.1


main :: IO ()
```

```
main = do
    dl <- getData "wdbc.csv"
    let (testS,trainS) = mkTestTrainSets p (splitClasses [0,1] dl)

    -- Get the length of the feature vector
    let dlen = length $ fst $ testS!!0
    print $ "Input length: " ++ show dlen

    -- Initialise and train network.
    let untrainedNN = [initLayer 6 dlen, initLayer 1 dlen]
    let trainedNN = trainNetwork 0.25 trainS 1000 untrainedNN

    -- Test and print (see definition below)
    print "Untrained network, test data:"
    testPrint untrainedNN testS
    print "Untrained network, training data:"
    testPrint untrainedNN trainS
    print "Trained network, test data:"
    testPrint trainedNN testS
    print "Trained network, training data:"
    testPrint trainedNN trainS

-- Auxiliar function.
testPrint n s = do
    let result = testNetwork n s
        -- testNetwork n s tests the classifier n on the data set s
        -- and returns a list of Boolean, indicating which data
        -- were correctly classified
        --
    let right = length [ x | x <- result, x ]
    let wrong = length [ x | x <- result, not x ]
    print ( "Correct: " ++ show right   )
    print ( "Errors:  " ++ show wrong )
```

1.2 Stack space overflow

It sometimes happens that you get a run time error, saying «Stack space overflow». This is often caused by Haskell's lazyness. Haskell will only evaluate an expression when the value is needed. All the unevaluated expressions must be stored, and sometimes the available stack space is filled up by unevaluated expressions. Especially recursive calls may under certain circumstances accumulate an exessive number of lazily unevaluated expressions.

1.2.1 Increasing stack space

There are runtime options to increase the stack space. If your program is called `test`, you can for instance run

```
./test +RTS -K10M
```

to increas the stack space to 10Mb.

You may experiment with different stack space sizes, but be aware that if you make it very large, the system may become very slow to respond because your program completely fills up the physical memory.

1.2.2 Strict evaluation

Increasing stack space is often useful, but very often we find that algorithms which should be able to run in constant stack space regardless of program size, in fact requires more and more stack space as parameter values increase. Then we need another solution.

As mentioned, the problem is with lazy, or more correctly non-strict, evaluation. A strict function will always evaluate its arguments. A non-strict function, which is the norm in Haskell, does not necessarily evaluate its arguments.

It is possible to make functions strict in one or more arguments. For example like this

```
{-# LANGUAGE BangPatterns #-}

foobar x !y = ...
          where
            !z = ...
```

The first line is a so-called pragma which is used to enables language extensions. The explanation mark is also known as a bang, and the notation using the explanation mark to force strictness is a bang pattern. Here, `y` is made strict, so a call to `foobar` will always force evaluation of the second argument. Similarly, there is a strict binding of `z` which means that it too is always evaluated.

Good use of bang patterns is not quickly learnt. You will have to look for other literature and examples when you need it.