Week 3: The Perceptron

Hans Georg Schaathun

25th April 2017

Time	Topic	Reading
8.15-	Recap of tutorials last week	
	Lecture: Classification and the Perceptron	Marsland Chapter 1-3
9.00-	Tutorial 3.1: The Perceptron	
11.45-	Lunch break	
12.15-	Recap/discussion of Tutorial 3.1	
	Lecture: I/O and compiled programs	Thompson Chapter 8
13.00-	Tutorial 3.2: I/O in Haskell	

This PDF document is available in an HTML version at `http://www.hg.schaathun.net/FPIA/week03.html`. Note that there are numerous external hyperlinks which you miss if you read a printed version of the document.

1 Tutorial 3.1: The Perceptron

Reading: Stephen Marsland: Chapter 1-3

In this tutorial we shall implement our first learning algorithm, namely a single neuron. The celebrate artificial neural networks (ANN) are built up of numerous neurons, so this tutorial is the first step.

1.1 Problem 1: Single Neuron and Perceptron Training

In this Problem we will implement only a single neuron with the perceptron training algorithm. The neuron is depicted in Figure 1. It acts as a function, with input $\vec{x} = (x_1, \ldots, x_n)$ on the left, and output y on the right. The weights $\vec{w} = (w_0, \ldots, w_n)$ defines a particular neuron. Different neurons (as far as we are concerned) use the same summing and thresholding functions, but they have different weights.

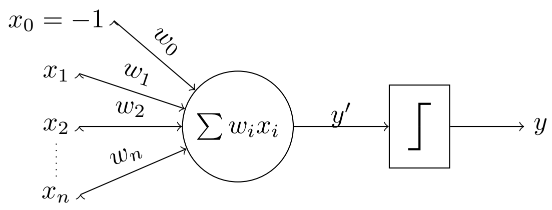

Figure 1: The single neuron with threshold function.

1.1.1 Step 1: Data Types

At the conceptual (mathematical) level, the neuron receives a real vector as input. The output is 0.0 or 1.0, which we also consider as a real number.

Discuss the following:

1. What data type should be used in Haskell for the output y?
2. What data type should be used in Haskell for the input $\vec{x}$?
3. What data type should be used in Haskell for the weights $\vec{w}$?

1.1.2 Step 2: The Neuron Type

Let us define a data type (alias) `Neuron` to represent a single neuron, recording all the weights.

1. Discuss: *What information must be stored as part of the neuron?*
2. Discuss: *What types can we use to define the* `Neuron` *type?*
3. Create a new module with the name `Perceptron`.
4. Add a definition for the `Neuron` type to the module.

1.1.3 Step 3: Initialisation

We need a function to create a new, pristine neuron. In a production system, this should be done randomly, but randomness is non-trivial, so we have to return to that later. For the time being, we are happy to initialise the neuron with small constant weights (say 0.01).

1. Give the type declaration for the initialisation function in your module:

```haskell
initNeuron :: Int -> Neuron
```

   The input is the number n of input values. The number of weights is $n + 1$.

2. Add a definition for `initNeuron`. The return value is a list of $n+1$ numbers, each equal to 0.01.

   You can start with the list `[0..n]` to get the right number of weights, and then use either `map` or list comprehension to generate a list of the same length and the right values (0.01).

3. Test the function in `ghci`. Does the function give you what you expect?

1.1.4 Step 4: Recall

The neuron as depicted in Figure 1 defines a function called *recall*. In Haskell it would have the following signature.

```
recall :: Neuron -> [Double] -> Double
```

The function takes the neuron and the input list, and it produces a scalar output.

1. Looking at Figure 1, we see that `recall` is the composition of two functions: the summation (circle) and the thresholding (square). In Haskell, this can be written as follows:

   ```
   recall :: Neuron -> [Double] -> Double
   recall n = threshold . neuronSum n
   ```

   It remains to implement the the `threshold` and `neuronSum`.

2. The threshold function is defined as

   $$\texttt{threshold}\ x = \begin{matrix} 0 & \text{for } x < 0, \\ 1 & \text{for } x \geq 0. \end{matrix} \qquad (1)$$

   Implement this function in your module using guards. Use the following type declaration.

   ```
   threshold :: Double -> Double
   ```

3. Secondly, we implement the summation (the circle node in Figure 1). Add the following to your module.

   ```
   neuronSum :: Neuron -> [Double] -> Double
   neuronSum n is = sum $ zipWith (*) n ((-1):is)
   ```

   Discuss how this function works.

   a) What does `(-1):is` mean?

   b) What does the `zipWith` function do?

   c) What does the `sum` function do?

4. Test the function. Start `ghci` and try the following

```
recall (initNeuron 3) [ 1.0, 0.5, -1.0 ]
```

Do you get the expected output?

5. Obviously, you do not learn all that you want to know from the above test, but at least you get to check for type errors. Develop your own test, by manually defining a test neuron with other weights, and use that in lieu of `initNeuron`.

1.1.5 Step 5: Training

The first step of implementing training is to update the neuron weights based on a single input/output pair. That is a function

```
trainOne :: Double -> [Double] -> Double
                -> Neuron -> Neuron
```

The first argument is the training factor η. The second is the input vector $\vec{x}$, and the third argument is the target output value t. The last (fourth) argument is the old neuron $\vec{w}$. The output is the updated neuron $\vec{w}'$.

The updated weights are defined as

$$w'_i = w_i - \eta(y - t)x_i, \text{ where} \tag{2}$$
$$y = \texttt{recall}\ \vec{w}\vec{x}. \tag{3}$$

In other words, if the actual output is different from the target output ($y \neq t$), then the weight is adjusted proportionally to the difference ($y - t$).

1. Implemente the weight update as defined above. We need a function with the following signature

```
weightUpdate :: Double -> Double -> Double
                       -> Double -> Double
weightUpdate eta diff w x  =
```

We have introduced `diff` for the different $y - t$, the other arguments are η, w and x as in (equation2). Complete the implementation and add it to your module.

2. We implement the `trainOne` as follows:

```
trainOne :: Double -> [Double] -> Double
                -> Neuron -> Neuron
trainOne eta xs t ws = zipWith (weightUpdate eta diff)
                               ws ((-1):xs)
                         where diff = recall ws xs - t
```

This implementation uses `zipWith` and partial application of `weightUpdate`. Discuss the following:

a) What does `zipWith` do?

b) What do we mean by partial application?

c) What is the type of the first argument to `zipWith`, i.e. `weightUpdate eta diff`?

3. To test the function, start `ghci` and try the following

```
trainOne 0.5 [ 1.0, 0.5, -1.0 ] 1.0 (initNeuron 3)
```

Do you get the expected output?

1.1.6 Step 6: Training on a Set of Vectors

The `trainOne` function uses only a single object for training. Now we need a `trainSet` function which uses a set of objects for training. This is a beautiful application of recursion over the list of training objects.

1. The function declaration is similar to that of `trainOne`, except that we get a lists instead of a single input vector and a single output value. Add it to your module as follows:

```
trainSet :: Double -> [[Double]] -> [Double]
                   -> Neuron -> Neuron
```

2. Add the base case:

```
trainSet _ [] _ n = n
```

Discuss, what does the base case do?

3. Then, add the recursive case:

```
trainSet eta (v:vs) (t:ts) n = trainSet eta vs ts
                             $ trainOne eta v t n
```

Discuss the following

a) What does the notation `(v:vs)` (and `(t:ts)`) mean?

b) What is the last argument to `trainSet`? What is its type? How is it computed?

c) How does the recursion work?

4. Test the function as you did with `trainOne`, but replace the input vector with a list of two input vectors (of you choice), each of length 3.

1.1.7 Step 7: Complete Training Function

It is usually not sufficient to run the training once only. Usually, we want to repeat the `trainSet` operation T times. In other words, we want a function with the following signature:

```
train :: Int -> Double -> [[Double]] -> [Double]
                 -> Neuron -> Neuron
```

The first argument is the number T of iterations, while the other argumens are as they were for `trainSet`.

1. Add the function declaration to your module.
2. Add a base case, defining the return value for $T = 0$ iterations.
3. Add a recursive case which uses `trainSet` to do a single iteration and calls `train` recursively to do $T - 1$ more iterations.

   You may look at the definition of `trainSet` above for an example of recursion, but remember that `train` recurses over an integer (the number of iterations) while `trainSet` recursed over a list.
4. Test the function using the same test data as you used for `trainSet`. Try both $T = 2$ and $T = 5$. replace the input vector with a list of two input vectors (of you choice), each of length 3.

1.1.8 Step 8: Testing

A simple test to device is to take a simple function and try to predict whether it is positive or negative. Take for instance the following:

$$f(x, y, z) = \texttt{threshold}\ \left[(x - 1.0)^4 + (y - 1.0)^5 - z - 1.0\right].$$

1. Choose a couple of feature vectors (x, y, z) (randomly or otherwise), and calculate the corresponding class label $f(x, y, z)$. This gives a training set.
2. Use the training set to train a neuron n in GHCi.
3. Choose another feature vector (x, y, z) and calculate $f(x, y, z)$. Use GHCi to calculate `recall` $(x, y, z)n$.
4. Compare the real class label $f(x, y, z)$ with the prediction obtained in GHCi. Do they match?
5. Repeat the last two steps a couple of times.

1.2 Problem 2: Multi-Neuron Perceptrons

1.2.1 Step 1: Data Type

Define a data type `Layer` *to represent a multi-neuron perceptron.*

What data type can be used to hold a set of neurons?

The name 'layer' will make sense when we advance to more complex neural networks. The perceptron consists of a single layer only, but other neural networks will be lists of layers.

1.2.2 Step 2: Initialisation

Define a function `initLayer` *to return a perceptron (layer) where all weights in all neurons is set to some small, constant, non-zero value.*

Remember arguments so that the user can choose both the number of neurons in the layer and the number of inputs. Each neuron in the layer should be created by a call to `initNeuron` which you defined above.

1.2.3 Step 3: Recall

Define a function `recallLayer` *which does a recall for each neuron in the layer, and returns a list of output values.*

1.2.4 Step 4: Training

Generalise each of the training functions `trainOne`*,* `trainSet`*, and* `train` *for perceptrons. The training functions for perceptrons have to apply the corresponding training function for each neuron in the layer.*

1.3 Epilogue

You have just implemented your first classifier. Well done.

However, this prototype leaves much to be desired.

1. We cannot initialise with random weights.
2. We have to type in the data for training and for testing.
3. We only have a single layer, and not a full network.

As you can see, we have to go back and learn some more techniques in Haskell. First of all, we will learn I/O in the next tutorial, to be able to read complete data sets from file, both for training and for testing.

2 Tutorial 3.2: I/O in Haskell

2.1 Overview

Reading: Simon Thompson: Chapter 7

In this tutorial we shall use real data to test the perceptron algorithm. In order to do this, we need to be able to read files from disk in a Haskell program.

2.2 Problem 1: Your first compiled program

2.2.1 Step 1: an output function

1. Try the following two evaluations in `ghci`

   ```
   "Hello World!"
   putStr "Hello World!"
   ```

   What is the difference between the two? Why is there a difference?

2. What are the types of the two expressions above? Do you know? Try it out using the `:type` command and see if it matches your expectation:

   ```
   :type "Hello World!"
   :type putStr "Hello World!"
   ```

The `IO ()` type is an example of a *monad*, a concept which will take some time to get used to. For the time being, we will only be concerned with the IO monad and how to use it to control I/O. We will learn more about monads later.

`IO` is a type constructor, so it wraps another type. In the case above, we had `IO ()`, with `()` as the inner type. This is the singleton type; i.e. the type `()` has only one possible value, namly `()`. What use can we have of singleton type?

The `IO` can be viewed as an action. Thus the type stores an action which can be subject to calculations and used to construct other actions. When the program runs, the action will eventually be performed.

Output actions, such as the one returned by `putStr`, will typically have type `IO ()`. They are interesting because of the output they generate, not because of the data contained. An input function, in contrast, could have type (say) `IO String` where the type wraps the data (string) read from input.

2.2.2 Step 2: sequencing

A program, typically, is a sequence of actions (e.g. IO objects). The easiest way to construct a program is the syntactic sugar of the `do` notation.

1. Create a new Haskell module called `Main` for this exercise.
2. Add the following definition:

```haskell
hello :: IO ()
hello = do
   n <- getLine
   putStr ( "Hello, " ++ n ++ "\n" )
```

   Note that we use two functions above, `getLine` and `putStr`.

3. What type does `putStr` have? Use the `:type` command if you do not know.
4. What type does `getLine` have? Use the `:type` command if you do not know.

   The `IO ()` type is just an action, with no contents. The `getLine` function returns an action with contents, and the `<-` operator *assigns* this contents to `n`.

5. Load your `Main` module in GHCI and evaluate `hello`. When nothing happens and you don't get a prompt, it is waiting for your input.
6. Type your name (or whatever), finish with Enter. What happens?

2.2.3 Step 3: compilation

The interpreter (`ghci`) is great to test individual functions, but at the end of the project you will probably want to produce a stand-alone program. This requires a compiler, namely `ghc`.

A standalone program is a module called `Main` with a function `main :: IO a` for some type `a`.

1. Add a main function to your `Main` module.

```haskell
main = hello
```

2. Save your module, and find a terminal window. Do not start GHCi. Compile your main module with the following command.

```
ghc Main.hs
```

3. List the contents of the directory

```
ls
```

(The Windows equivalent to `ls` is `dir`.) Which new files have been created?

4. Run the resulting program on the command line, as follows:

```
./Main
```

(On Windows you may need to run `Main.exe` instead of `./Main`.) What happens?

It is possible to get GHC to make programs with names other than Main, but let's cross that bridge when we need it.

2.3 Problem 2: Reading a data set

We want to test our machine learning algorithm on real data. University of California, Irvine hosts the machine learning repository which provides a large collection of real data for testing. We will use some breast cancer data from Wisconsin.

2.3.1 Step 1: What does the data look like?

1. Have a brief look at the details about the data set. What kind of information is available?
2. Download the data file.
3. Move the data file to the directory you use for this tutorial.
4. Open the data file in your text editor (the same as you use to write Haskell code).
5. Discuss: How is the data formatted? Where do you find the class label?
6. Discuss: Which data types are used in the data set?

Comma separated values (CSV) is a common format to store data. Each row is a record, and each item of the record is separated by commas. We need to figure out how to read such files in Haskell.

2.3.2 Step 2: Reading a text file

In the previous step we download a file with comma-separated values (CSV), which we want to use with our perceptron. Let's explore how we can read the file in Haskell.

1. Make sure you have the data file wdbc.data in your current directory, and start GHCi.
2. Run the following in expression:

```
readFile "wdbc.data"
```

What do you get?

2.3.3 Step 3: Installing a library

To parse the CSV file, we will use a library which is not installed by default. Hackage is a database of libraries for Haskell, and you are likely to consult it frequently for new libraries. We shall take a brief look at Hackage and the documentation found there.

1. Look up the Text.CSV library. The first page gives an overview.
2. Look at the list of modules. Once you have installed the library, these modules are accessible with the `import` statement in Haskell.

   Which modules are available?
3. Click on the `Text.CSV` module. This gives the API documentation for this module. Which types and functions can you use? (Don't spend too much time on this if you don't see the answer. We will walk through together.)
4. Look at the header line of the web page, in the top left corner. This is the package name: `csv`. To install the package, you have to find a terminal window (do not start GHCi) and run the following command:

   ```
   cabal update
   cabal install csv
   ```

2.3.4 Step 4: Testing the CSV library

As yoe see in the API documentation, the CSV library has several functions to parse CSV data. Since we have already learnt how to read the file into a String, we will use the function `parseCSVTest` which parses a String.

1. Find a terminal window and start `ghci`.
2. Import the CSV module

   ```
   import Text.CSV
   ```
3. Lets define a String object with CSV data.

   ```
   let s = "1,2,3\n4,5,6"
   ```
4. The `parseCSVTest` function takes one argument, namely the CSV formatted string. Try this

```
parseCSVTest s
```

Look at the output. What data type is returned?

5. What is the return type of `parseCSVTest`? You can check the documentation or use GHCi with the following command.

```
:type parseCSVTest
```

Discuss: Does this type make `parseCSVTest` suitable in a program?

2.3.5 Step 5: Parsing CSV from a string

The `parseCSVTest` is a test function which prints the data on the terminal. It does not actually return the data. To be able to use the data for further computation, we will use `parseCSV`.

1. What is the return type of `parseCSV`?

2. There are two 'kinds' of objects of the `Either` type. Try the following in GHCi:

```
:type Left 'a'
:type Right 2
```

The `Either` type allows us to pack two constituent types (the left and the right type) into one. We can use an `Either` object without knowing which constituent type is used.

3. The return type of `parseCSV` is either a 'Left' which means it is a ParseError, or 'Right' which means it is a valid CSV object.

Doscuss: Why doesn't `parseCSV` just return `CSV`? What is the `ParseError` for?

4. In production software you have to take care of ParseError to do error handling. However, there is a simple and crude fix to convert the `Either` object to a plain `CSV` object. We will make a function for this.

Create a new module called `ANNData` and add the following definition.

```
stripError :: Either a b -> b
stripError (Left _) = error "Parser error!"
stripError (Right csv) = csv
```

Discuss the following:

a) How is pattern matching applied to objects of the `Either` type?

b) What does `a` and `b` mean in the type declaration?

c) What does the `error` function do?

5. Test the `stripError` function in GHCi. Do for instance:

```
stripError (Left "foobar")
stripError (Right 3.14)
```

6. Discuss: What does the `error` function do?

7. The first argument to `parseCSV` is the name of a log file. We won't use that for now, so let's write a simple wrapper for `parseCSV`. Add the following to the `ANNData` module:

```
parseCSVsimple :: String -> CSV
parseCSVsimple s = stripError (parseCSV "/dev/null" s)
```

Here, `/dev/null` is a special file discarding all data written thereto. (The special file does not exist on Windows, and Windows users may have to use a real file instead.)

8. Test `parseCSVsimple` in the GHCi, in same way as you tested `parseCSVTest`.

2.3.6 Step 6: Parsing a real CSV file

We have learnt to read a file into a string, and to parse a string for CSV data. Now, we will put these two operations together and make a function to read and parse a real data set from file.

1. Add the following type declaration to the `ANNData` module.

```
getRawData' :: String -> IO [[String]]
```

The input argument is the filename from which the data will be read. The output is a list of lists, where each constituent list is one row from the CSV file, and each string in the inner list is one value from the comma separated line.

2. We implement `getRawData'` as follows:

```
getRawData' fn = do
                  s <- readFile fn
                  return $ parseCSVsimple s
```

The `return` function wraps the given value in an `IO` action.

Discuss: What is the meaning of the `<-` operator?

3. Test the function `getRawData'` on the Wisconsin Breast Cancer Data file.

```
getRawData' "wdbc.data"
```

What output do you see? Does it fit you expectation?

Remark 1 *There is a slightly simpler way to do this. You can make a wrapper similar to* `parseCSVsimple`*, using* `parseCSVFromFile` *instead of* `parseCSV`*. Try it out for yourself if you have time.*

2.3.7 Step 7: A little problem with real CSV data

It is possible that the data from `parseCSVsimple` includes an empty row, `[""]`.

1. Write a function `dropEmpty` which takes a list of lists, as returned by `getRawData'`, and drops any list containing just the empty string, and keeping all others.

   Add both type declaration and definition to the `ANNData` module.

2. Define the following function

```
getRawData :: String -> IO [[String]]
getRawData fn = do
                d <- getRawData' fn
                return (dropEmpty d)
```

2.3.8 Step 8: Cleaning up the data

So far we have read and parsed the data set to obtain a list of lists of strings. However, the data are numerical, so String is not an appropriate data type. We need to clean it up, and parse the strings containing numbers into a numeric data type.

Each row in the CSV file includes several values which would form the input vector to a perceptron, plus a class which determines the the correct output.

1. Look at the «attribute information» in the presentation of the data set, as well as the data file. What is the meaning of the individual columns? Which are input? Which is output?

Cleaning up the data is a multi-step process, which we consider in the next problem.

2.4 Problem 3: Cleaning up the data

The data set (CSV) file consists of rows. Each row consists of an ID, a class label, and a feature vector. The feature vector is in turn made up of individual features.

The raw data that you have read is [[String]], so each row is a list of strings, where one string is class label, some strings may be ignored (the ID), and the rest is the feature vector.

We want to reformat the data set so that it has type [(Double,[Double])]. Thus each row is a pair, where the first element is the class label (Double) and the other is the feature vector ([Double]). Thus, we need the function

```
formatData :: [[String]] -> [(Double,[Double])]
```

It is easiest to work bottom up. So we will do formatData last, and start with the class label and individual features.

2.4.1 Step 1: Formatting the class label

The class label is a string "M" or "B", while it should be numeric, typically 0 or 1. Let's map "M" to 1 and "B" to 0. We need a function `numericLabel` to do the conversion

1. Write a type declaration for `numericLabel`
2. Write an implementation for `numericLabel`
3. Test the function

   ```
   numericLabel "M"
   numericLabel "B"
   numericLabel "q"
   numericLabel "Bonnie"
   ```

For the time being, it is ok if the last two tests cause an error. In a production system we would have to handle such errors appropriately. Our time, in contrast, is better spent on exploring the learning algorithm, than handling input which we do not want to see.

2.4.2 Step 2: Formatting the feature vector

The features are strings representing numeric data. We have to parse it to get floating point data. We need a function `numericFeatures` to do the conversion.

1. We need `read` function to do the conversion. Open `ghci` and get familiar with it. Try the following:

   ```
   read "6.12"
   ```

   What happens?

2. You get a rather cryptic error message. What it essentially says is that GHCI does not know which data type you want for the return value. You have to specify this explicitely. Try the following:

   ```
   read "6" :: Integer
   read "6" :: Double
   read "6.12" :: Double
   ```

   What happens now?

3. Write the type declaration for `numericFeatures`.
4. We can define `numericFeatures` using `map` and `read` as follows:

   ```
   numericFeatures = map read
   ```

If you have a precise type declaration for `numericFeature`, GHCi can deduce the return type required from `read`; thus you do not need to specify the type again.

5. Test the function

```
numericFeatures ["6.12","8.11","0","2"]
numericFeatures ["B","6.12","8.11","0","2"]
```

For the time being, it is ok if the last test causes an error. As before, a production system would require adequate error handling.

2.4.3 Step 3: Formatting the record

Using the helper functions from Steps 1-2, we are ready to write a function `processItem` taking a row ([String]) from the parsed CSV data and return a pair with class label and feature vector for the perceptron.

1. Write a type declaration for `processItem` in the `ANNData` module.
2. Add a function definition for `processItem`, using the helper functions from Steps 1-2.
3. Test the function, e.g.

```
processItem ["9898","M","6.12","8.11","0","2"]
```

2.4.4 Step 4: Formatting the complete data set

Now we need a function `formatData` taking [[String]] as input and applying `processItem` on each row. The output should be a list of class label/feature vector pairs. This is an obvious case for `map`.

1. Write a type declaration for `formatData`.
2. Write a definition for `formatData`.
3. Test the function on data from the `getRawData` function.

2.4.5 Step 5: Putting it all together

Now, at last, we can make a single `getData` function which does it all. Starting with file name as input, it reads the file, parses CSV data, and formats it properly using `formatData`.

1. Write a type declaration for `getData` in the `ANNData` module.

2. Using all the functions you have implemented above, add a definition of the `getData` function.

3. Test the `getData` function on the breast cancer data set in GHCi. Are you happy with the output?

2.5 Problem 4: Refinement (optional)

As you see in the API documentation, the CSV library has several functions to parse CSV data. The one we used is very simple and provides no error handling.

Revise the functions above to use `parseCSV`, and handle error values properly.