Week 6: Pseudo-Random Number Generators

Hans Georg Schaathun

25th April 2017

Time	Topic	Reading
8.15-	Recap of tutorials last week	
	Lecture: Pseudo-Random Number Generators	
9.00-	Tutorial 6.1:	
11.45-	Lunch break	
12.15-	Questions and answers	
	Lecture: Monads and State machines	Thomson Chapter 18
13.00-	Tutorial 6.1 continued	

This PDF document is available in an HTML version at `http://www.hg.schaathun.net/FPIA/week06.html`.

1 Tutorial 6.1: Random weights

In this tutorial we amend the initeialisation function so that we use random starting weights.

Keep a copy of your original program, so that you can compare performance.

1.1 Step 1: The `tf-randm` package

We are going to use the PRNG provided by the `tf-randm` package.

1. First, check that you have installed the package

   ```
   cabal install tf-random
   ```

2. Run GHCi and test the functions we are going to use:

```
import System.Random
import System.Random.TF
g <- newTFGen
g
```

What kind of object is `g`?

3. Continue testing:

```
next g
```

What is the return value of `g`?

What is the type of `next`? (Use the `:type` command to check.)

4. What is the `Word32` type? Use google or hoogle to look it up if you do not know.

1.2 Step 2: A random floating point value

We want to generate a list of random floating point value, i.e. Double. All the library functions provide integers, so we need to make our own function for this.

You choose whether you want to create a new module for this tutorial, or add the definitions to your neural network module.

1. To make definitions using the `Word32` data type, you need the following import in your module:

```
import Data.Word
```

2. The `Word32` type contains integers in the range $0 \ldots 2^{32} - 1$. What mathematical function can you use to map this range to a real number in the $[0, 1]$ or $[0, 1)$ range.

3. Make a Haskell function which maps a `Word32` to a `Double` between 0 and 1.

```
toZeroOne :: Word32 -> Double
```

4. The random weights should be in the range $(-\alpha, +\alpha)$ for some small value of α. Make a function which takes an α value and a `Word32` value and produces a number in the $(-\alpha, +\alpha)$ range.

```
toWeight :: Double -> Word32 -> Double
```

You may use the above `toZeroOne` if you want to.

1.3 Step 3: A random list

For each neuron, we need a list of weights. In Haskell, it is easiest to generate an infinite list and then use the `take` function. First we define the following function to generate the infinite list:

```
rndWeights :: Double -> TFGen -> [Double]
```

The first input is the bound α as defined in the previous step.

Implement this function in whatever way you want. If you cannot see how, use the following hints.

1.3.1 Hints

1. Write a recursive function which takes a `TFGen` generator as input and produces a list of `Word32`.

   ```
   rndList :: TFGen -> [Word32]
   ```

   Use recursion. You do not need a base case here. Why not?

2. Define the `rndWeights` function using `map`, `rndList`, and `toWeight`.

1.4 Step 3: A random neuron

Now we can define a new `initNeuron` function. Let's call it `initNeuronG` since it takes a generator as input..

```
rndNeuron :: TFGen -> Integer -> (Neuron,TFGen)
```

The second input is the number of inputs to the Neuron.

To implement this function, you need to:

- use `split` to get a new generrator for the `TFGen` output.
- use `rndWeights` to produce the random weights for the `Neuron` output.
- use the `take` function (look it up if you do not know it) and to get the right number of weights in the `Neuron`.
- make sure that you do not use the same generator twice.

1.5 Step 4: A random layer

Using all the techniques from Steps 2–3, you can now make a recursive function to generate a random layer of neurons.

1.5.1 Optional Step 4bis

Make a function to generate a random multi-layer network.

1.6 Step 5: Putting it into your program

All the functions we have used so far are pure. They are fully deterministic with the generator as input.

In the `main` function of your program, you have acess to the `IO` monad, and thus to `newTFGen`.

1. Copy the test program you already have for your perceptron. (If you are confident in what you are doing, you may skip the perceptron and jump straight to the multi-layer network,)
2. In the copy, you can define you starting generator using

   ```
   g0 <- newTFGen
   ```

3. Change the initialisation of the network (or perceptron) to use the pseudo-random functions that you have defined above.
4. Have you ever used the same generator twice? Make sure that you do not.
5. Repeat this step with a test program for your multi-layer network.

2 Tutorial 6.2: Testing

Test both your versions, with fixed and with random weights, three times each and compare the results. What observations can you make?