CS 103 Computer Programming

Assignment Number 8*

April 11, 2014

Deadline: 18th April, 2014 before 23h45

Attention

- Make sure that you read and understand each and every instruction. If you have any questions or comments you are encouraged to discuss your problems with your colleagues (and instructors) on Piazza.
- Plagiarism is strongly forbidden and will be very strongly punished. If we find that you have copied from someone else or someone else has copied from you (with or without your knowledge) both of you will be punished. You will be awarded straight zero in this assignment or all assignments.
- Your code must be properly documented, encapsulated and should avoid any form of duplication.

Brain Modeling We are interested in this exercise to build a very basic model of brain consisting of a set of neurons (*c.f.* Figures 2 and 5). A brain neuron has following characteristics:

- A neuron is connected to other neurons;
- A neuron receives an external signal from all the neurons it is connected to (separately), and transmits it, optionally modified, to all those which are connected with it;
- Certain neurons may be of specialized types.

Description Either download the program `neuronesTest.cpp` or copy paste the snippets given in Figures 1 and 6 and complete the missing parts so that this test code works properly.

Required Basic Tools To complete the task you will need some basic tools this includes writing a `Point` class to store the coordinates of a two dimensional Cartesian point. Your `Point` class must provide the necessary overloaded functions required to manipulate and display a 2D point and must also provide a `distance()` function to compute Euclidean distance between any two 2D points. Recall that Euclidean distance between two points is: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Now lets move to the main part.

1. A Basic Neuron Your basic Neuron class should mimic a basic neuron and must be characterized by following properties:

- a 2D position (`position_`).
- a signal of type double (`signal_`).which corresponds to the response of neuron to the received stimulus from outside.
- an attenuation factor of signal of type double (`attenuation_`).
- a set of "neighbours" neurons (`neighbours`) which will be connected with it and to whom it will send the signals.

Please also note that a single neuron may well have been connected to several different neurons (*i.e.* it can receive information from several neurons).

In short, your Neuron class must at least provide the following functionality:

- The class constructor will initialize the position and the attenuation factor `(default=1)` using values passed as parameters (see the main() supplied); the received signal will be zero and all the neurons to which it is connected will be empty initially.
- Accessor `position() and signal()` methods for attributes (`position_ and signal_`).

*Errors and Omissions expected.

```cpp
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
// Testing Code.
int main ()
{
  // Test of  part 1
  cout << "===== Test of 1st part  =====" << endl << endl;

  Neuron neuron1(0, 1, 0.5);
  Neuron neuron2(1, 0, 1.0);
  Neuron neuron3(1, 1, 2.0);

  neuron1 += &neuron2;
  neuron1 += &neuron3;
  neuron2 += &neuron3;

  cout << neuron1 << endl;

  neuron1.fire(10);
  cout << "Signals :" << endl;
  cout << neuron1.signal() << endl;
  cout << neuron2.signal() << endl;
  cout << neuron3.signal() << endl;

  // Test of  part 2
  cout << endl << "===== Test of 2nd part =====" << endl << endl;

  CumulativeNeuron neuron4(0, 0, 0.75);
  cout << neuron4 << endl;
  neuron4.fire(10);
  neuron4.fire(10);
  cout << "Signal :" << endl;
  cout << neuron4.signal() << endl;

  //Copy Paste the test code of Logical Gate OR
  cout << "===== Test of Logical gate \"or\"   =====" << endl << endl;

  // Add the code for other logical gates..
  cout << "===== Test of Logical gates \"And, etc\"   =====" << endl << endl;


 // Copy paste the Test Code for xnor gate here.
  cout << "===== Test of Logical gate \"xnor\"   =====" << endl << endl;

  return 0;
}
```

Figure 1: Example code for testing the program.

- It should be possible to connect a new Neuron (via a pointer) using the "+=" operator, as shown in the main() provided
- A `fire((double received_signal)` method will stimulate the neuron via an external signal. This method will simply call two auxiliary methods (possibly polymorphic) in this order.
  1. `void accumulate(double received_signal)` that calculates the internal signal to the neuron. In its basic version here, it sets the attribute `signal_` to the product of the signal received and attenuation factor.
  2. `void propagate()` that calls the `fire` method of each of the "neighbours" (connected) neurons and pass its `signal_`'s value as argument, *i.e.* the current neuron signal is propagated to all the children.
- A method `display()` to display the information of the neuron as shown in the example sequence, typically:

```
The neuron at position (0, 1) with an attenuation factor of 0.5  is
connected to following (x) neuron(s):
   - Neuron at position (1, 0)
   - Neuron at position (1, 1)
```

  If a neuron is not connected to any other neuron, then display should print the message "not connected to any neuron".

It should also be possible to display the neurons using the usual ostream operator. Finally, please note that since neurons cannot be copied neither assigned to another neuron, it should not be possible to either copy or make any assignment of Neuron object.

You can test the program implemented so far, using the section "Test Part 1" of the provided `main()` function.

2. Specialized neurons There are different types of specialized neurons. We will cover here the `CumulativeNeuron`. These neurons:

1. First accumulate the signals received (accumulated using the `accumulate` method) from all the connected neurons instead of simply storing the last received signal, *i.e.*

$$signal\_ = signal\_ + received\_signal_i$$

2. Next they propagate the *clipped and attenuated version of accumulated signal* to all the connected neurons. Note that `propagate` method is called only once when signals from all the connecting neurons is received. The clipped and attenuated version of accumulated signal will be computed using following relation[1]:

$$signal\_ = \left(\frac{1}{1 + exp(-signal\_)}\right) \times attenuation$$

For the proper functionality of CumulativeNeuron you *might* need to introduce two auxiliary variables: `total_incoming_connections` and `counter`; where `total_incoming_connections` will record the number of incoming neuron connections from which it can receive signals and `counter` will be used to (i) keep count of neurons from which signals have been received as well as to tell the number of still missing neurons from which signal is expected and (ii) will be used to call the `propagate` method; in the propagate method you will set this counter to zero once again so that signal counting can be restarted from incoming neurons.

You can test the program implemented so far, using the section "Test Part 2" of the provided `main()` function.

Some Simple Logical Gates Now since we have basic lets put them to the task to build simple artificial brains for solving some basic problems. We are going to build some simple logic gates. These logic gates can be build using combination of basic neurons and a CumulativeNeuron (*c.f.* Figure 2 – Notice that using very a simple architecture we can build these gates).

Let's start writing the code for "or (|)" logical gate. To build this architecture we will build three basic neurons with appropriate attenuation weights and finally connect them with a CumulativeNeuron. Figure 3 shows the building and testing code for "or" logical gate . Note that to get the output from the CumulativeNeuron we will call its display function which will display its final computed signal.

Follow the similar lines to add the code for the remaining three logical gates.

[1]This function is usually called sigmoid(logistic) activation function

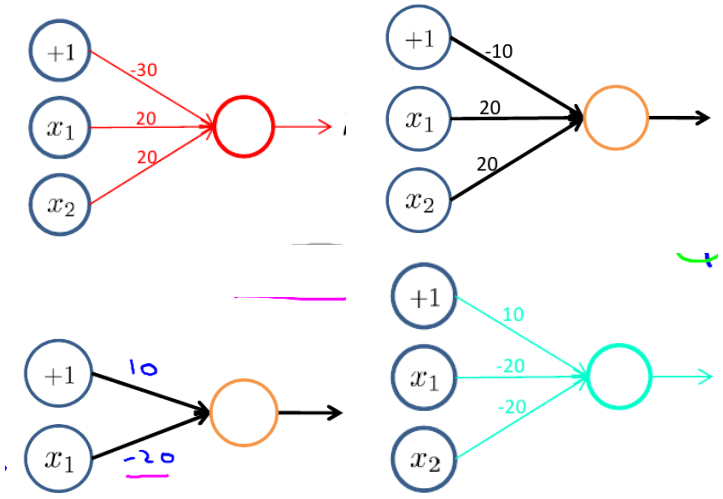

Figure 2: Different organizations of Neurons for building logic gates. First row shows the neuron organizations for building "and (a & b)" (left) and "or ($a \mid b$)" (right) logic gates. Second row shows the neuron organizations for building "not ($\sim a$)" (left) and "($\sim a$ & $\sim b$)" (right) logic gates. *Note that here in each diagram circles on the left represent basic neurons with arrows depicting there attenuation weights and circles on right represent the CumulativeNeuron.*

Complex Systems Now the questions arises can we build all types of logical gates (such as xor or xnor, *etc.*) using these neurons ? Unfortunately, the answer is no. Our these neurons are too simple and basic as they provide the same signal to all the connected neurons (because the input signal is attenuated by same amount) so then what ?

Luckily, we can update simple versions of our basic and cumulative neurons to make them more powerful.

Complex Neurons To model complex systems we will be using complex versions of basic and cumulative Neurons. Figure 4 highlights the differences between simple and complex neurons based architectures – blue circles in the figure represent the basic neurons and orange ones represent cumulative neurons. Note that *the only difference* between these two architectures is that simple versions of basic and cumulative neurons (*e.g.* in Figure 4 basic neuron x_1 uses same attenuation factor a_1 for both connected x_3 and x_4 cumulative neurons) use the same attenuation factor for all the connected neurons. In comparison, the complex versions of these neurons use different attenuation factor for each connected neuron (*e.g.* in Figure 4 x_1 uses attenuation factor of a_{13} and a_{14} respectively for connected x_3 and x_4 neurons).

Now your goal is to write the complex versions of the Neuron and CumulativeNeuron classes. Note that since now each connected neuron will have its own attenuation factor we can no longer use the "+=" operator for attaching a neuron to another neuron with constant attenuation factor. So your both classes must provide the `attach(double, Neuron*)` function for attaching a neuron with its respective attenuation weight. Moreover, your ComplexNeuron class should provide the specialized version of `fire` method. This method will call the `fire` method of connecting neurons and pass its signal weighted by respective attenuation factor as an argument, *e.g.* to call fire method of i_{th} neuron it will use something such as `fire(`$signal\_ \times attenuation_i$`)`. Your ComplexCumulativeNeuron `fire` method will call its `accumulate` method first to accumulate the signals from all the incoming neurons and once it has received signals from all the incoming neurons it will compute the "clipped version of accumulated signal" and weight it by corresponding attenuation factor before propagating to the connected neuron.

Building Complex Models Using our new complex neurons based brains now we can model any complex system ranging from all types of logical gates to system for face detection, recognition, *etc.* (of course if we are given the true attenuation factors [2]).

Now lets use our these complex form of neurons to build the "xnor" gate. We will be using the architecture shown in the Figure 5. For building this system we will need four complex neurons and three complex cumulative neurons. First we build all the neurons objects and connect them among themselves to complete the architecture. Finally, we provide the input to x_1 and x_2 via `fire` method. The `fire` method of neuron transmits the weighted signal to each connected neuron. Complex cumulative neurons receive the signals and keep log of the fact that from how many neurons they need to receive signals before to propagate the attenuated clipped version of their signal to their outgoing neurons by calling the `propagate` method. The testing code is given in the Figure 6. Now your goal is provide the missing code for this task.

[2] Note that Google, Facebook, *etc.* use similar systems for finding friends, pages, detecting and recognizing faces, cars, humans, *etc.*, where the attenuation factors are learned automatically from the users labelled data — this data is provided by users via tagging of different online content. Here for the sake of simplicity we are omitting the details of algorithms used to automatically learn the ways. For curious minds, these algorithms involves nothing more than simple partial derivatives and matrix operations :) and can be implemented quite easily :-D

```
// Test of Logical OR Gate
cout << "===== Test of Logical gate \"or\"   =====" << endl << endl;
// building the architecture...
// Here n1 and n2 will be our inputs, n0 will be always 1.
Neuron n0(2, 0, -10);
Neuron n1(1, 0, 20.0);
Neuron n2(0, 0, 20.0);

CumulativeNeuron n3(1, 3, 1); // will act as output

n0+=&n3;
n1+=&n3;
n2+=&n3;

// Now lets check the system.
n0.fire(1); // will always fire 1...

n1.fire(0)
n2.fire(1);
cout<< "If a=0 and b=1 then a|b == " << n3; //

n1.fire(1)
n2.fire(1);
cout<< "If a=1 and b=1 then a|b == " << n3; //

n1.fire(0)
n2.fire(0);
cout<< "If a=0 and b=0 then a|b == " << n3; //
```

Figure 3: Code used to build architecture of "or" logical gate.

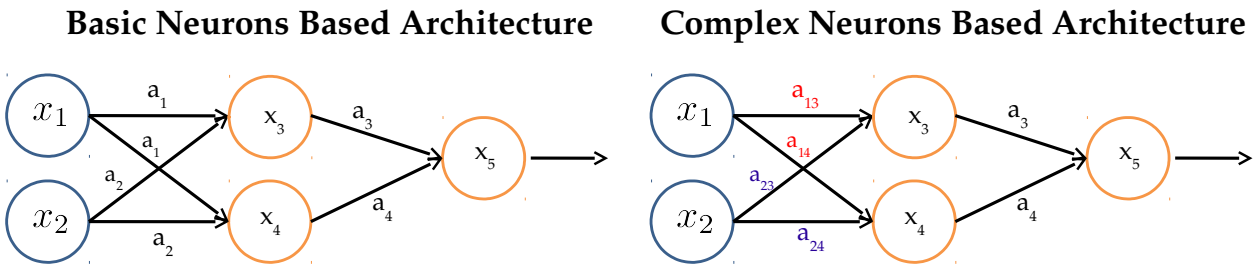

Figure 4: Comparison between architectures based on simple and complex neurons. Notice that the difference between the simple (left) and complex (right) versions is only due to difference in the attenuation factors being used by the complex neurons. *Note that here in each diagram circles on the left represent basic neurons with arrows depicting there attenuation weights and circles on right represent the CumulativeNeuron.*

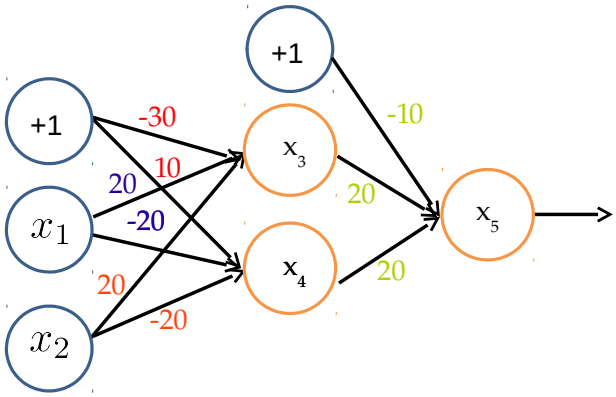

Figure 5: Xnor model built using complex versions of neuron. Here x_1 and x_2 are input neurons and x_5 is output neuron. Note that since $a \ \ xnor \ \ b \ = \ (a \ \& \ b) \mid (\sim a \ \& \sim b)$ the xnor is build the architecture of same gates (*i.e. c.f.* Figure 2)

```
  // Test  of Logical XNOR gate...
ComplexNeuron x0(2, 0);
ComplexNeuron x1(1, 0); // x1 as input
ComplexNeuron x2(0, 0); // x2 as input


ComplexCumulativeNeuron x3(1, 3, 1);
ComplexCumulativeNeuron x4(0, 3, 1);
ComplexNeuron x6(3, 3, 1);


ComplexCumulativeNeuron x5(6, 2, 1);

// layer-1 neurons

x0.attach(-30,&x3);
x0.attach(10,&x4);

x1.attach(20,&x3);
x1.attach(-20,&x4);

x2.attach(20,&x3);
x2.attach(-20,&x4);


// layer-2 neurons

x6.attach(-10,&x5);

x3.attach(20,&x5);
x4.attach(20,&x5);

x0.fire(1);
x6.fire(1);

// Lets start checking the system by calling the fire method of input neurons
x1.fire(0)
x2.fire(1);
cout<< "If a=0 and b=1 then ~(a^b) == " << x5; //

x1.fire(1)
x2.fire(1);
cout<< "If a=1 and b=1 then ~(a^b) == " << x5; //


x1.fire(0)
x2.fire(0);
cout<< "If a=0 and b=0 then ~(a^b) == " << x5; //
```

Figure 6: Code used to build architecture of “xnor” logical gate.