Agile Artificial Intelligence

ALEXANDRE BERGEL

2017-12-13

Contents

1 History

Humans have always been fascinated by an intelligence emerging from something else than Human. Vikings have myth of monsters made of stones and metal.

Modern Artificial Intelligence began with the invention of programmable digital computer in the 40s. In 1948, Alan Turing designed “Machine Intelligence” as machine automatically solving problems.

Arthur Samuel, in his 1983 talk titled “AI: Where it Has Been and Where It is Going” said that the main goal of machine learning and artificial intelligence is *to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence.*

2 Artificial Intelligence

“One of the central challenges of computer science is to get a computer to do what needs to be done, without telling it how to do it.” – John Koza

3 Agile Artificial Intelligence

About bringing agility in the way techniques related to artificial intelligence are designed, implemented, and evaluated. This implies that artificial intelligence systems is built in an iterative way and using an incremental design.

Our code is often accompanied with unit tests and visualizations, and written in a programming environment supporting …

This book focus on implementing effective techniques that are commonly associated to Artificial Intelligence. Only a very small portion of what Artificial Intelligence is is covered in this book.

4 Why this book?

There exist many sophisticated libraries to build, train, and run neural networks and genetic algorithms. So, why this book?

Most of artificial intelligence techniques are often perceived as a black box that operates in a almost magical way. The purpose of Agile Artificial Intelligence is to reveals some of the most arcane algorithms. This book details how neural networks and genetic algorithms may be written from scratch, without any supporting libraries.

Understanding the machinery behind fantastic algorithms and techniques is a natural goal that one should pursue. This first allow one to (i) easily hook into if necessary, and (ii) understand the scope of these algorithms.

This book is meant to detail some easy-to-use recipes to solve punctual problems and highlights some technical details using the Pharo programming language.

5 Requirements to read this book

Agile Artificial Intelligence is designed to have a large audience. In particular, there no need to have knowledge in neural networks or genetic algorithm. There is even no need to have strong mathematical knowledge. However,

6 What if I am not a Pharo programmer?

The only way to salve you is to learn Pharo :) Pharo has a very simple syntax, which means that even for a Java programmer, code should be understandable. Chapter 2 also presents the Pharo programming language and environment.

7 Additional Readings

If you want to know more about some of the techniques presented here, I do recommand the following books: - http://www.gp-field-guide.org.uk by Riccardo Poli, Bill Langdon, and Nic McPhee - http://natureofcode.com by Daniel Shiffman - http://neuralnetworksanddeeplearning.com

If I want to know more about Pharo: - http://files.pharo.org/books/updated-pharo-by-example/ provides a nice and smooth introduction to Pharo. - http://agilevisualization.com describes the Roassal - http://files.pharo.org/books contains many books and booklets that cover Pharo

Perceptron

This chapters plays two roles. The first one, is to describes how and why a perceptron plays a role so important in the meaning of deep learning. The second role of this chapter, is to provide a gentle introduce to the Pharo programming language.

8 Biological Connection

The primary visual cortex contains 140 millions of neurons, with tens of billions of connections. A typical neuron propagates electrochemical stimulation received from other neural cells using *dendrite*. An *axon* conducts electrical impulses away from the neuron (Neuron).

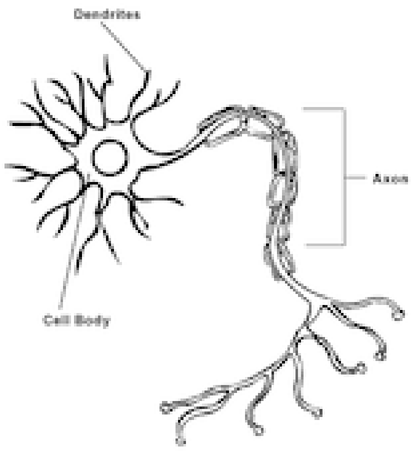

Figure 1: *Neuron*

Expressing a computation in terms of artificial neurons was first thought in 1943, by Warren S. Mcculloch and Walter Pitts in their seminal article *A logical calculus of the ideas immanent in nervous activity*. This paper has a significant impact in the field of artificial intelligence. It is interesting to realize the knowledge we had about neurons at that time.

9 Perceptron

A perceptron is a kind of artificial neuron that models the behavior of a real neuron. A perceptron is a miniature machine that produces an output for a provided input (Perceptron). A perceptron may accept 0, 1, or more inputs, and output the result of a small and simple computation. A perceptron operates on numerical values, which means that the inputs and the output are numbers (integer or float, as we will see later).

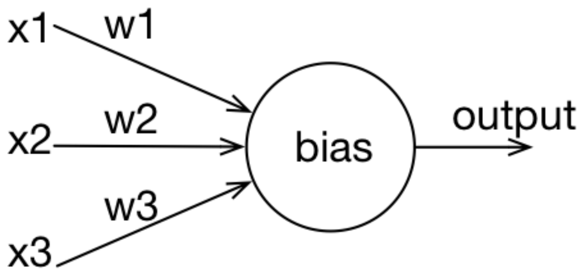

Figure 2: *Perceptron*

The figure depicts a perceptron with three inputs, noted *x1*, *x2*, and *x3*. Each input is indicated with an incoming arrow and the output with the outgoing arrow. The $y = xˆ2 when $x > 2$ $ $\sum 12a^2 + b^2 = c^2$

Not all inputs have the same importance for the perceptron. For example, an input may be more important than the others. Relevance of an input is expressed using a weight associated to that input. In our figure, the input *x1* has the weight *w1*, *x2* has the weight *w2*, and *x3* has *w3*.

How likely is the perceptron responding to the input stimulus? The bias is a value that indicates whether

10 A Perceptron in action

We have seen so far a great deal of theory. We will implement a perceptron from scratch.

We first need to open a code *system browser* by selecting the corresponding entry in the World menu. The system browser is where you read and write source code. Most of the programming activity will actually happens in a system browser. A system browser is composed of five different parts. The above part is composed of four lists. The left-most provides the packages, the following list gives the classes that belongs to a selected package. The third list gives the method cateogies for the selected class. A method category is a bit like a package, but for methods. The left-most list gives the methods that belongs in the class under a particular method category. The below part of a system browser gives source code, which is either a class template to fill in order to create a class, the source code of the selected class, or the source code of a selected method.

Right-click on the left-most top list to create a new package, let's call it `NeuralNetwork`. This package will contain most of the code we will write in this book.

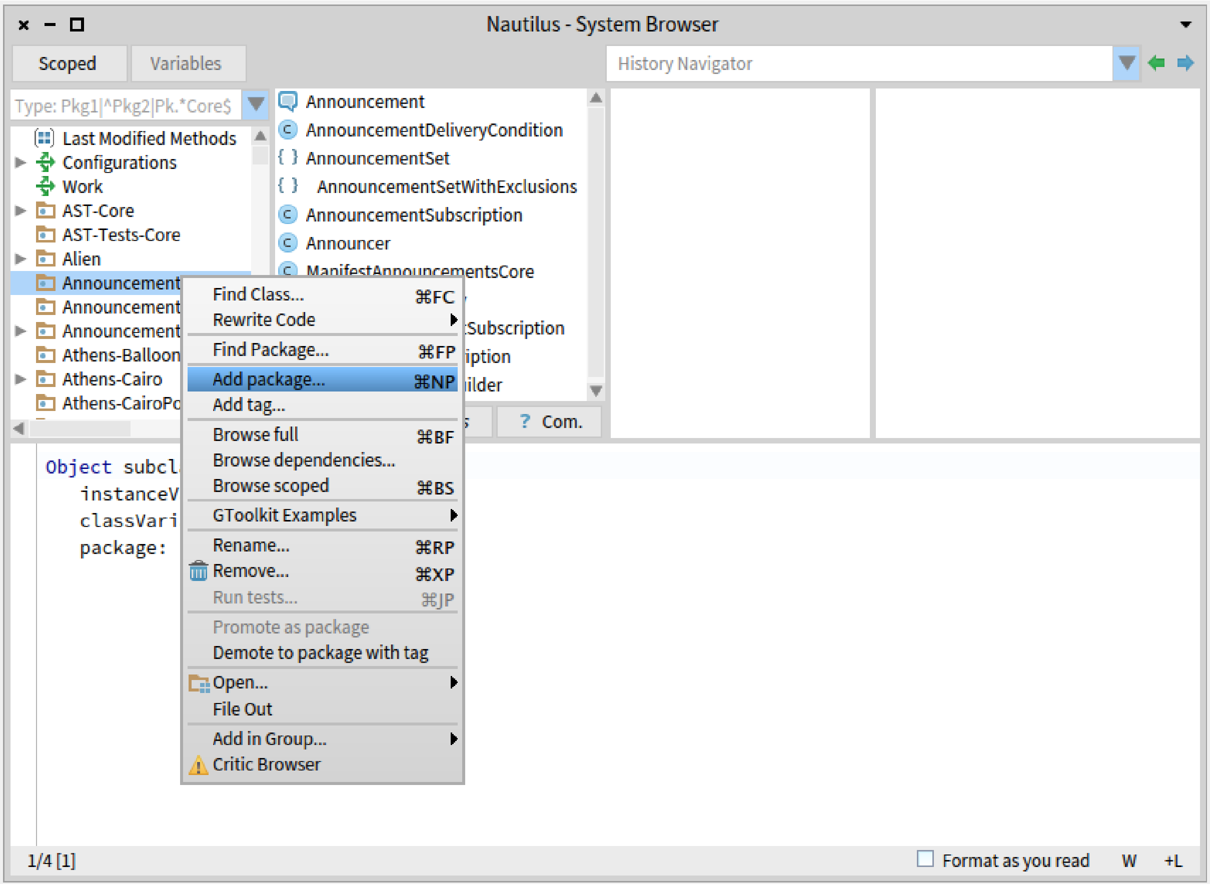

Figure 3: *systemBrowser*

When you select our package, a class template appears in the below code. Fill it to have the following:

```
Object subclass: #Neuron
    instanceVariableNames: 'weights bias'
    classVariableNames: ''
    package: 'NeuralNetwork'
```

You then need to compile the code by "'accept"'-ing the source code. Right click on the text pane and select the option""Accept"'. The class we have defined contains two instance variables, `weights` and `bias`. We now have to add a few methods that manipulate these variables before some actual work. Let first focus on manipulating the `weights` variable. We will define two methods to write a value to that variable and another to read from it.

You may wonder why we define a class `Neuron` and not `Perceptron`. In the future chapter we will turn our class `Neuron` into an open abstraction to artificial neuron.

Here is the code of the `weights:` method defined in the class `Perceptron`:

```
Neuron>>weights: someWeightsAsNumbers
    weights := someWeightsAsNumbers copy
```

To define this method, you need to select the class in the class panel (second top list panel). Then write the code given above *without* `Neuron>>`. Then you should accept the code, by right clicking on the "Accept" menu item. Accepting a method has the effect to compile it. The code define the method named `weights:` which accepts one argument, provided as a variable named `someWeightsAsNumbers`.

The expression `weights := someWeightsAsNumbers copy` creates a copy of the provided argument and assign it to the variable `weights`. The copy is not necessary, but it is useful to prevent some hard-to-debug issues.

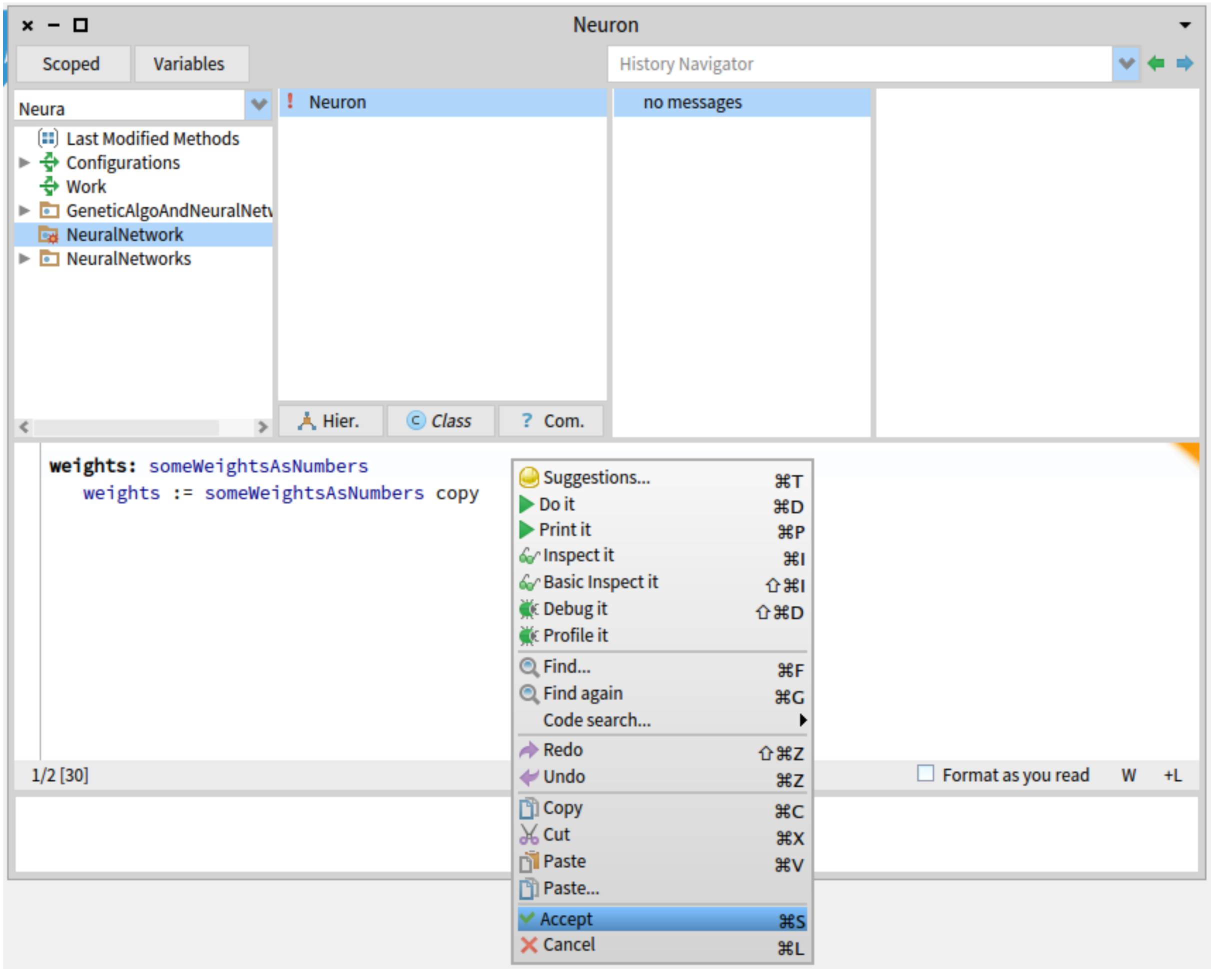

Figure 4: *systemBrowserAndMethodWeight*

We know need a method to read the content of that variable. Here is the apropriate method:

```
Neuron>>weights
    ^ weights
```

The character `^` returns the value of an expression, the value of the variable `weights` in that case.

Similarly we need to define a method to assign a value to the `bias` variable and to read its content. The method `bias:` can be defined as:

```
Neuron>>bias: aNumber
    bias := aNumber
```

And the reading may be defined using:

```
Neuron>>bias
    ^ bias
```

So far, we have defined the class `Perceptron` which contains two variables (`weights` and `bias`), and 4 methods (`weights:`, `weights`, `bias:`, and `bias`). The last piece to add is applying a set of inputs values and obtaining the output value. The method `feeds:` can be defined as:

```
Neuron>>feed: inputs
    | z |
    z := (inputs with: weights collect: [ :x :w | x * w ]) sum + bias.
    ^ z > 0 ifTrue: [ 1 ] ifFalse: [ 0 ].
```

The method `feed:` simply phrase the formula to model the activation of a perceptron into the Pharo programming language. The expression `inputs with: weights collect: [ :x :w | x * w ]` collects for each pair of elements (one from `inputs` and another from `weights`) using a function. Consider the following example:

```
#(1 2 3) with: #(10 20 30) collect: [ :a :b | a + b ]
```

The above expression evaluates to `#(11 22 33)`. Syntactically, it means that the literal value `#(1 2 3)` receives a message called `with:collect:`, with two arguments, the literal `#(10 20 30)` and the block `[ :a :b | a + b ]`. You can verify the value of that expression by opening a playground, accessible from the World menu. A playground is a kind of command terminal ("'e.g.,"' xterm in the Unix World).

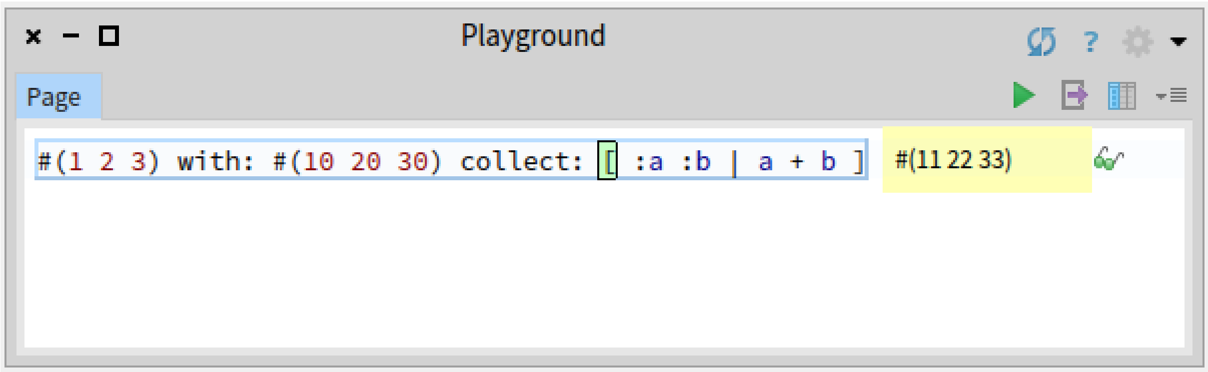

Figure 5: *playground*

We can now play a little bit with a perceptron. Evaluate the following code in a playground:

```
p := Neuron new.
p weights: #(1 2).
p bias: -2.
p feed: #(5 2)
```

If you do the math, this piece of code evaluates to `1` (since `(5*1 + 2*2)- 2` equals to `7`, which is greater than `0`).

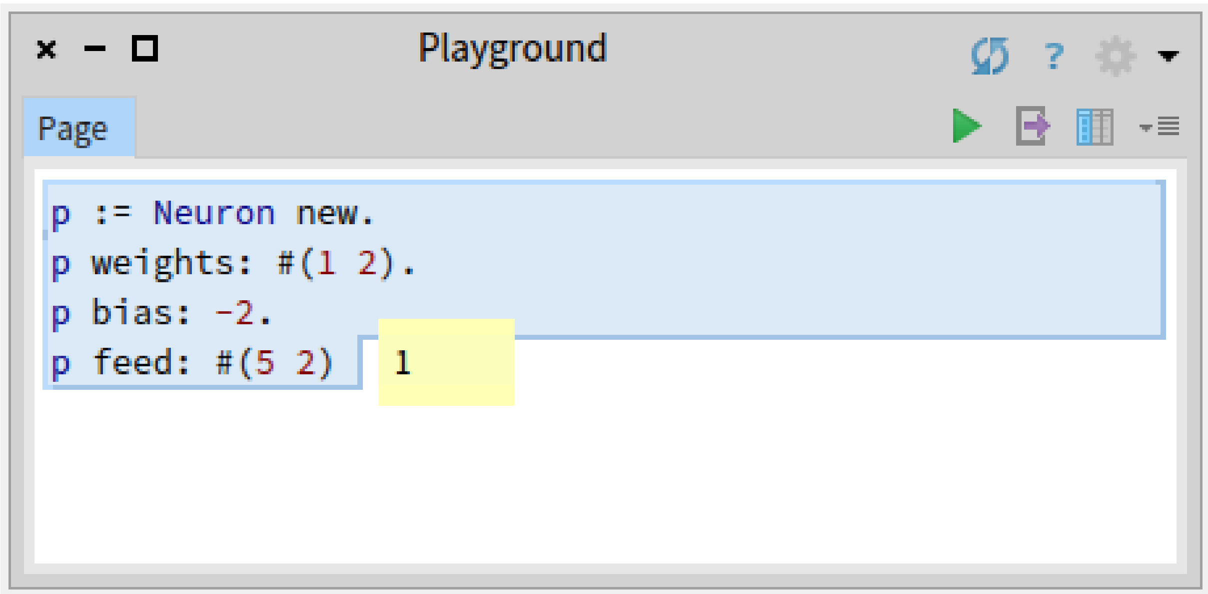

Figure 6: *playgroundAndPerceptron*

11 Testing our code

Now is time to talk about testing. Testing will be a pillar of our activity of code production. Software and code testing is essential in agile methodologies and is about raising the confidence that the code we write does what it is supposed to do.

Testing is a central concept that we will borrow from the field of Software Engineering. Although this book is not about writing large software artifact, we *do* produce code. And making sure that this code can be tested in an automatic fashion significantly help improve the quality of what we are doing. More importantly, it is not only the author of the code (you!) that will appreciate the quality of the code, but anyone who will look at it. Along the chapters, we will improve our codebase. It is therefore very important to make sure that our improvement do not break some of the functionalities.

For example, above we defined a perceptron, and we informally tested it in a playground. This informal test costed us a few keystrokes and a little bit of time. What if we can automatically repeat this test each time we modify our definition of perceptron? This is exactly what *unit testing* is all about.

We now define a class called `PerceptronTest`, defined as:

```
TestCase subclass: #PerceptronTest
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'NeuralNetwork'
```

The class `TestCase` belongs to the Pharo codebase and subclassing it is the first step to create a unit test. Tests can now be added to our `PerceptronTest`. Define the following method:

```
PerceptronTest>>testSmallExample
    | p result |
    p := Neuron new.
    p weights: #(1 2).
    p bias: -2.
    result := p feed: #(5 2).
    self assert: result equals: 1.
```

The test can be run by clicking on the gray circle located next to the method name.

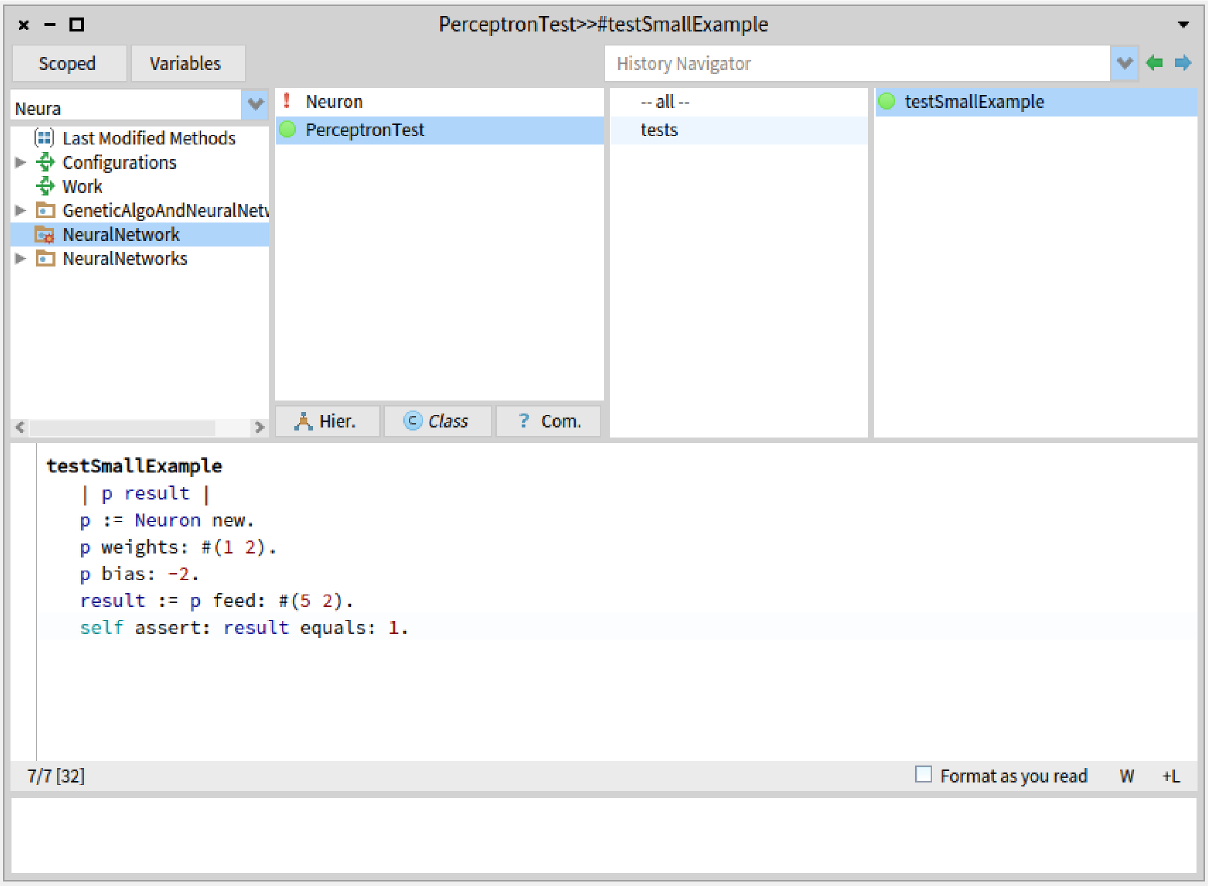

Figure 7: *testingPerceptron01*

A green color indicates that the test passes (*i.e.,* no assertion failed and no error got raised). The method `testSmallExample` sends the message `assert:equals:` which tests whether the first argument equals the second argument.

EXERCISE: So far, we have only shallowly tested our perceptron. We can improve our tests in two ways: - In `testSmallExample`, test that feeding our perceptron `p` with differents values (*e.g.,* −2 and 2 gives 0 as result) - Test our perceptron with different weights and bias

In general, it is a very good practice to write a good amount of tests, even for a single component unit as for our class `Perceptron`.

12 Formulating Logical expressions

A canonical example of using perceptron (or any other artificial neuron) is to express boolean logical gates. The idea is to have a perceptron with two inputs (each being a boolean value), and the result of a logical gate as output.

A little bit of arithmetic indicates that a perceptron with the weights `#(1 1)` and the bias `-1.5` formulates the AND logical gate. We could therefore verify this with a new test method:

```
PerceptronTest>>testAND
    | p |
    p := Neuron new.
    p weights: { 1 . 1 }.
    p bias: -1.5.

    self assert: (p feed: { 0 . 0 }) equals: 0.
    self assert: (p feed: { 0 . 1 }) equals: 0.
    self assert: (p feed: { 1 . 0 }) equals: 0.
    self assert: (p feed: { 1 . 1 }) equals: 1.
```

Similarly, a perceptron can formulate the OR logical gate. Consider the following test:

```
PerceptronTest>>testOR
    | p |
    p := Neuron new.
    p weights: { 1 . 1 }.
    p bias: -0.5.

    self assert: (p feed: { 0 . 0 }) equals: 0.
    self assert: (p feed: { 0 . 1 }) equals: 1.
    self assert: (p feed: { 1 . 0 }) equals: 1.
    self assert: (p feed: { 1 . 1 }) equals: 1.
```

Negating the weights and bias results in the negated logical gate:

```
PerceptronTest>>testNOR
    | p |
    p := Neuron new.
    p weights: { -1 . -1 }.
    p bias: 0.5.

    self assert: (p feed: { 0 . 0 }) equals: 1.
    self assert: (p feed: { 0 . 1 }) equals: 0.
    self assert: (p feed: { 1 . 0 }) equals: 0.
    self assert: (p feed: { 1 . 1 }) equals: 0.
```

So far we had perceptron with two inputs. A perceptron accepts the same number of inputs than the number of weights. Therefore, if only one weight is provided, only one input is required. Consider the NOT logical gate:

```
PerceptronTest>>testNOT
    | p |
    p := Neuron new.
    p weights: { -1 }.
    p bias: 0.5.

    self assert: (p feed: { 1 }) equals: 0.
    self assert: (p feed: { 0 }) equals: 1.
```

13 Combining Perceptrons

So far, we have defined the AND, OR, and NOT logical gates. A combination of these gates may produce a digital comparator circuit, illustrated as:

The circuit compares the value of input A and B. We have possible three outcomes: - A is greater than B - A is equal to B - A is lesser than B

We can therefore model our circuit with two inputs and three outputs. The following table summarizes the circuit"

A	B	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

The circuit is defined as:

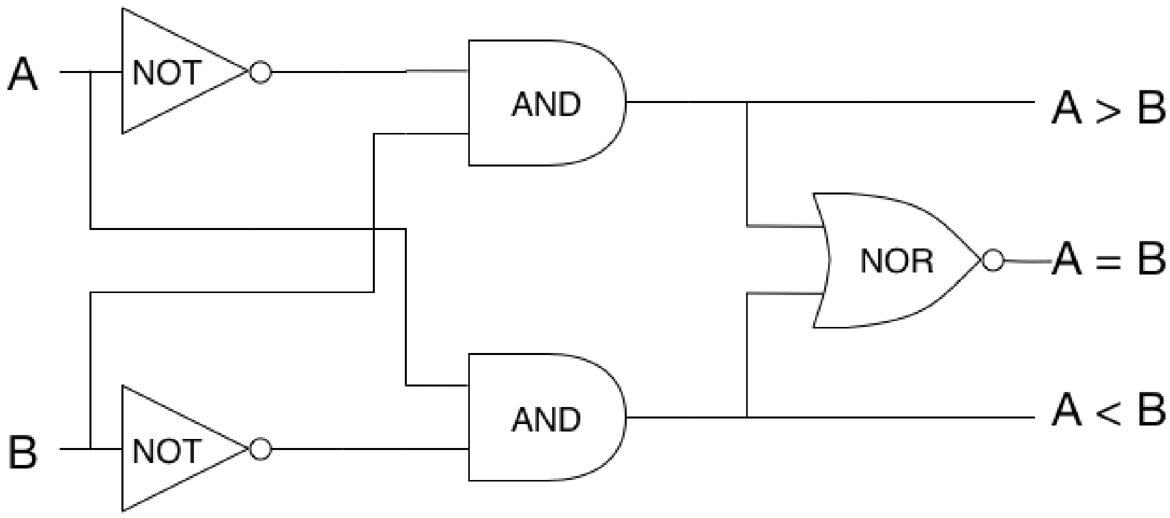

Figure 8: *digitalComparator*

Three logical gates are necessary: AND, NOT, and NOR. We then need to make the connection between these gates. As we previously did, some tests will drive our effort. The method `digitalComparator:`, defined in our unit test for convenience, models the digital comparator circuit:

```
PerceptronTest>>digitalComparator: inputs
    "Return an array of three elements"
    | not and nor A B AgB AeB AlB notA notB |
    A := inputs first.
    B := inputs second.

    and := Neuron new weights: { 1 . 1 }; bias: -1.5.
    not := Neuron new weights: { -1 }; bias: 0.5.
    nor := Neuron new weights: { -1 . -1 }; bias: 0.5.

    notA := not feed: { A }.
    notB := not feed: { B }.

    AgB := and feed: { notA . B }.
    AlB := and feed: { A . notB }.
    AeB := nor feed: { AgB . AlB }.
    ^ { AgB . AeB . AlB }
```

The method accept a set of inputs as argument. We first extract the first and second elements of these inputs and assign them to the temporary variables `A` and `B`.

We then create our three logical gates as perceptrons. We then wire then using the variables `notA`, notB, AgB (standing for A greater than B), AlB (A lesser than B), and AeB (A equals to B). The method returns an array with the result of the circuit evaluation. We can test it using a test method:

```
PerceptronTest>>testDigitalComparador
    self assert: (self digitalComparator: { 0 . 0 }) equals: { 0 . 1 .
        0 }.
    self assert: (self digitalComparator: { 0 . 1 }) equals: { 1 . 0 .
        0 }.
    self assert: (self digitalComparator: { 1 . 0 }) equals: { 0 . 0 .
        1 }.
    self assert: (self digitalComparator: { 1 . 1 }) equals: { 0 . 1 .
        0 }.
```

We have now seen how perceptrons may be "manually" combined by using variables having a particular order of the evaluation (*e.g.,* the variable notA must be computed before computing an output). When we will discuss about training a neural network, we will come back to that particular example. A training will actually (i) compute some weights and bias and (ii) establish the wire between the neurons automatically.

14 Training a Perceptron

Making neurons learn is essential to make something useful. Learning typically involves a set of input examples with some known output. The learning process assess how good the artificial neuron is against the desired output. In particular, as defined by Frank Rosenblatt in the late 1950s, each weight of the perceptron is modified by an amount that is proportional to the product of the input and the difference between the real output and the desired output.

Learning in neural networks means adjusting the weights and the bias in order to make the output close to the set of training examples. Our way to train a perceptron will therefore has to adjust its weights and bias according to how good it performs for a given set of inputs.

A way to make perceptron learn is given by the method train:desiredOutput:, as follow:

```
Neuron>>train: inputs desiredOutput: desiredOutput
    | learningRate theError output |
    output := self feed: inputs.
    learningRate := 0.1.

    theError := desiredOutput - output.

    inputs withIndexDo: [ :anInput :index |
```

```
        weights at: index put: ((weights at: index) + (learningRate *
            theError * anInput)) ].

    bias := bias + (learningRate * theError)
```

Before doing any adjustment of the weights and bias, we need to know how well the perceptron evaluates the set of inputs. We therefore need to evaluate the perceptron with the argument `inputs`. The result is assigned to the variable `realOutput`. The variable `difference` represents the difference between the desired output and the real output. We also need to decide how fast the perceptron is supposed to learn. The `learningRate` value is a value between `0.0` and `1.0`. We arbitrarily picked the value `0.1`.

Let's see how to use the training in practice. Consider the perceptron `p` given as (you can evaluate the following code in a playground):

```
p := Neuron new.
p weights: { -1 . -1 }.
p bias: 2.
p feed: { 0 . 1 }.
```

As we have seen, we have `p feed: { 0 . 1 }` equals to `1`. What if we wish the perceptron to actually output `0` for the input `{ 0 . 1 }`? We therefore need to train `p` to actually output `0`. Let's try the following:

```
p := Neuron new.
p weights: { -1 . -1 }.
p bias: 2.
p train: { 0 . 1 } desiredOutput: 0.
p feed: { 0 . 1 }.
```

Evaluating this expression still outputs `1`. Well… Were we not supposed to train our perceptron? The learning process is a rather slow process, and we need to actually teach the perceptron a few times what the designed output is. We can repeatably train the perceptron as in:

```
p := Neuron new.
p weights: { -1 . -1 }.
p bias: 2.
10 timesRepeat: [ p train: { 0 . 1 } desiredOutput: 0 ].
p feed: { 0 . 1 }.
```

Evaluating the code given above produces `0`, as we were hopping for. Our perceptron has learned!

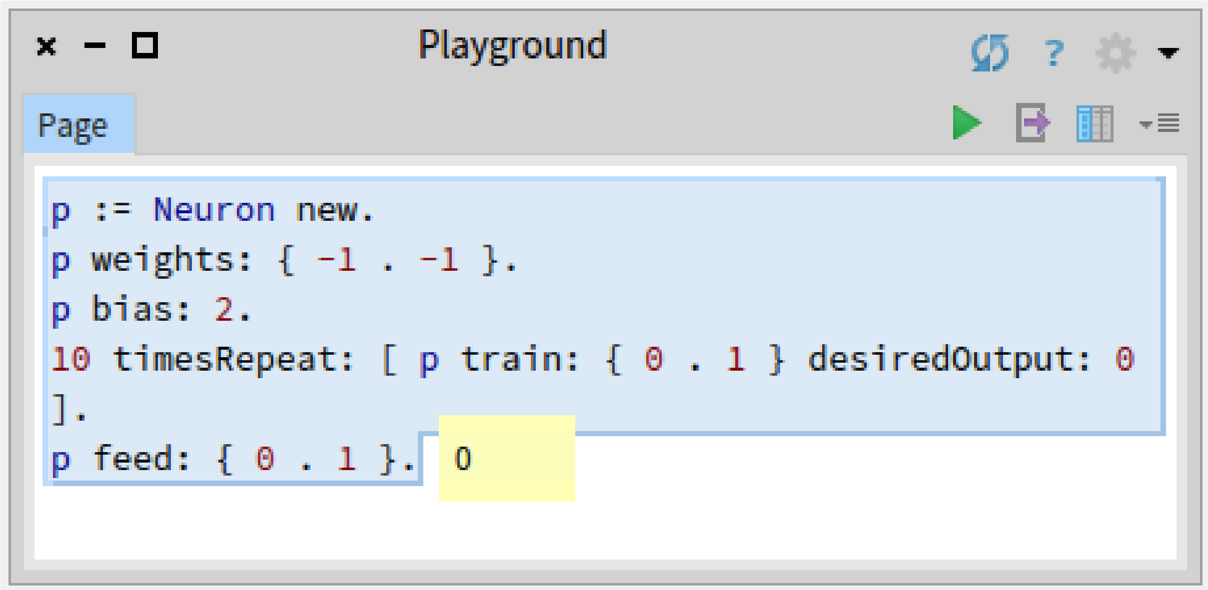

Figure 9: *playgroundWithLearningPerceptron*

We can now train some perceptron to actually learn how to express the logical gates. Consider the following `testTrainingOR`:

```
PerceptronTest>>testTrainingOR
    | p |
    p := Neuron new.
    p weights: { -1 . -1 }.
    p bias: 2.

    40 timesRepeat: [
        p train: { 0 . 0 } desiredOutput: 0.
        p train: { 0 . 1 } desiredOutput: 1.
        p train: { 1 . 0 } desiredOutput: 1.
        p train: { 1 . 1 } desiredOutput: 1.
    ].

    self assert: (p feed: { 0 . 0 }) equals: 0.
    self assert: (p feed: { 0 . 1 }) equals: 1.
    self assert: (p feed: { 1 . 0 }) equals: 1.
    self assert: (p feed: { 1 . 1 }) equals: 1
```

The method `testTrainingOR` first creates a perceptron with some arbitrary weights and bias. We successfully train it with the four possible combinations of the OR logical gate. After the training, we test the perceptron to see if it has actually properly learn.

In `testTrainingOR`, we train the perceptron 40 times the complete set of examples. Training a perceptron (or a large neural network) with the complete set of examples is called *epoch*. So, in our example, we train `p` with 40 epochs.

EXERCISE: - What is the necessary minimum number of epochs to train `p`? You can try to modify `25` by a lower value and run the test to see if it still passes. - We have shown how to train a perceptron to learn the OR logical gate. Write a method `testTrainingNOR`, `testTrainingAND`, and `testTrainingNOT` for the other gates we have seen. - How the value of the `learningRate` impacts the minimum number of epochs for the training?

15 Predicting side of a 2D point

A perceptron, even as the simple one we designed, can be used to classify data and make some predictions. Consider the following example:

- We have a space composed of red and blue points
- A straight line divides the red points from the blue points

Some questions arise:

- Can we teach a perceptron to correctly assign the color of a point?
- How many example points do we need to train the perceptron with in order to make good prediction?

Let's pick a linear function, such as $f(x) = -2x - 3$. A given point (x, y) is colored in red if $y > f(x)$, else it is blue. Consider the following script:

```
somePoints := OrderedCollection new.
500 timesRepeat: [
    somePoints add: {(50 atRandom - 25) . (50 atRandom - 25)}
].

f := [ :x | (-2 * x) - 3 ].

"We use the Grapher engine to plots our points"
g := RTGrapher new.
d := RTData new.
d dotShape
    color: [ :p | (p second > (f value: p first))
                    ifTrue: [ Color red trans ]
                    ifFalse: [ Color blue trans ] ].
d points: somePoints.
```

```
d x: #first.
d y: #second.
g add: d.
g
```

Inspecting this code snippet produces a graph with 500 colored dots.

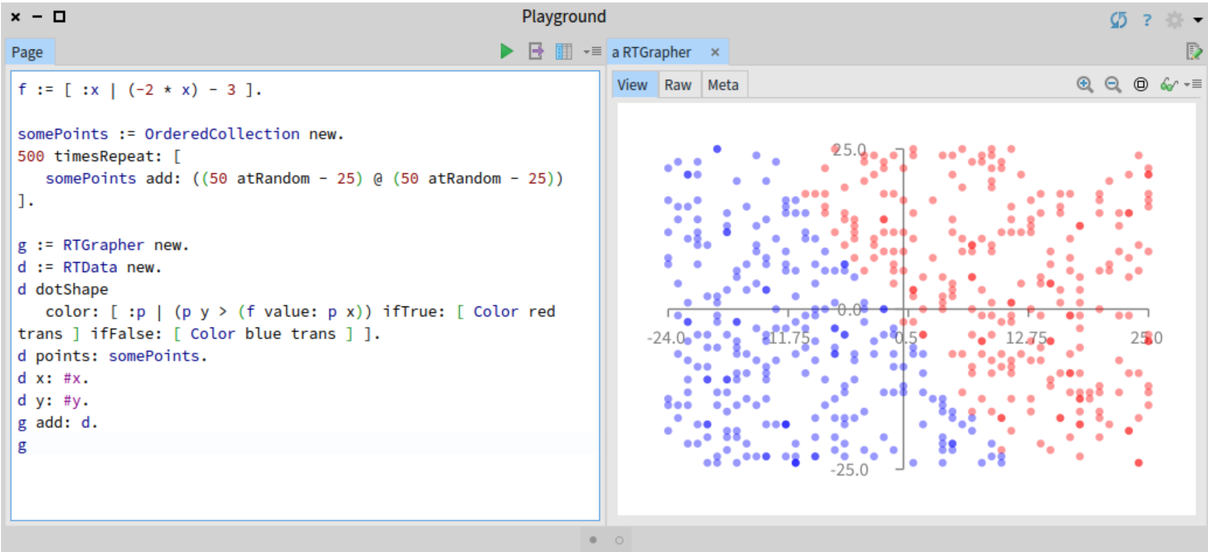

Figure 10: *simpleLine*

The script begins by defining a set of 500 points, ranging within a squared area of 50 (from -25 to +25). Each point is created by sending the message `atRandom` to the object number `50`. This message send return a number randomly picked between 1 and 50. Each point is represented as an array of two numbers. Our 500 points are kept in a collection, instanced of the class `OrderedCollection`.

We then assign the block to the variable `f`. The block corresponds to the function $f(x)$ written using the Pharo syntax. A block may be evaluated with the message `value:`. For example, we have `f value: 3` that returns `-9` and `f value: -2` that returns `1`.

The remaining of the script uses Grapher to plot the points. A point `p` is red if `p y` is greater than `f value: p x`, else it is blue. The color `Color red trans` indicates a transparent red color.

We can add the actual line defined by `f` in our graph. Consider the small revision:

```
somePoints := OrderedCollection new.
500 timesRepeat: [
    somePoints add: {(50 atRandom - 25) . (50 atRandom - 25)}
].

```

```
f := [ :x | (-2 * x) - 3 ].

g := RTGrapher new.
d := RTData new.
d dotShape
    color: [ :p | (p second > (f value: p first))
                    ifTrue: [ Color red trans ]
                    ifFalse: [ Color blue trans ] ].
d points: somePoints.
d x: #first.
d y: #second.
g add: d.

"Added code below"
d2 := RTData new.
d2 noDot.
d2 connectColor: Color red.
d2 points: (-15 to: 15 by: 0.1).
d2 y: f.
d2 x: #yourself.
g add: d2.
g
```

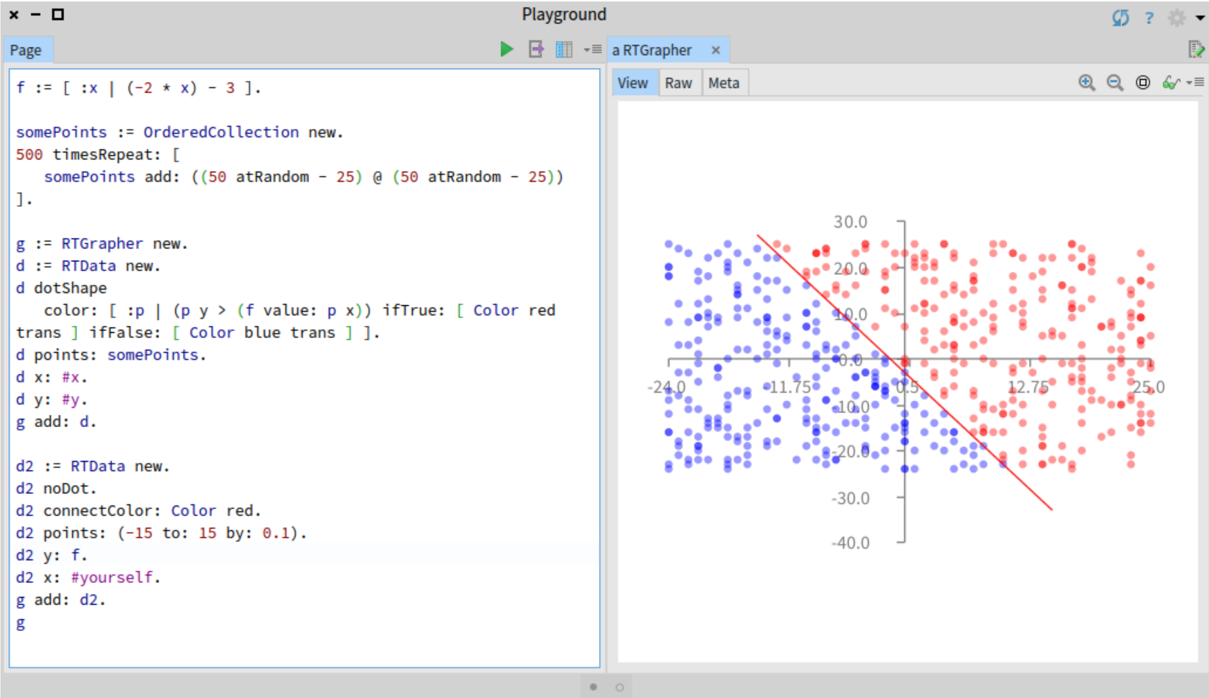

Figure 11: *simpleLine2*

We will now add a perceptron in our script and see how good it performs to guess on which side of the line a point is. Consider the following script:

```
f := [ :x | (-2 * x) - 3 ].
p := Neuron new.
p weights: { 1 . 2 }.
p bias: -1.
r := Random new seed: 42.

"We are training the perceptron"
500 timesRepeat: [
    anX := (r nextInt: 50) - 25.
    anY := (r nextInt: 50) - 25.
    designedOutput := (f value: anX) >= anY
                            ifTrue: [ 1 ] ifFalse: [ 0 ].
    p train: { anX . anY } desiredOutput: designedOutput
].

"Test points"
testPoints := OrderedCollection new.
2000 timesRepeat: [
    testPoints add: { ((r nextInt: 50) - 25) . ((r nextInt: 50) - 25) }
```

```
].

g := RTGrapher new.
d := RTData new.
d dotShape
    color: [ :point | (p feed: point) > 0.5
                ifTrue: [ Color red trans ]
                ifFalse: [ Color blue trans ] ].
d points: testPoints.
d x: #first.
d y: #second.
g add: d.

d2 := RTData new.
d2 noDot.
d2 connectColor: Color red.
d2 points: (-15 to: 15 by: 0.1).
d2 y: f.
d2 x: #yourself.
g add: d2.
g
```

As earlier, the script begins with the definition of the block function `f`. It then creates a perceptron with some arbitrary weights and bias. Subsequently, a random number generator is created. In our previous scripts, to obtain a random value between 1 and 50, we simply wrote `50 atRandom`. Using a random number generator, we need to write:

```
r := Random new seed: 42.
r nextInt: 50.
```

Why this? First of all, being able to generate random numbers is necessary in all stochastic approaches, which includes neural networks and genetic algorithms. Although randomness is very important, we usually not want to let such random value creates situations that cannot be reproduced. Imagine that our code behaves erratically for a given random value, that we do not even know. How can we track down the anomaly in our code. If we have truly random numbers, it means that executing twice the same piece of code may produce (even slightly) different behaviors. It may therefore be complicated to properly test. Instead, we will use a random generator with a known seed to produce a known sequence of random numbers. Consider the expression:

```
(1 to: 5) collect: [ :i | 50 atRandom ]
```

Each time you will evaluate this expression, you will obtain a *new* sequence of 5 random numbers.

Using a generator you have:

```
r := Random new seed: 42.
(1 to: 5) collect: [ :i | r nextInt: 50 ]
```

Evaluating several times this small script always produces the same sequence. This is key to have reproducible and deterministic behavior. In the remaining of the book, we will intensively use random generators.

Our script then follow with training a perceptron with 500 points. We then create 2000 test points, which will be then displayed on the screen, using Grapher. Note that we write the condition `(p feed: point)> 0.5` to color a point as red. We could have `(p feed: point)= 1` instead, however in the future chapter we will replace the perceptron with another kind of artificial neuron, which will not exactly produce the value 1.

We see that our the area of red points goes closely follows the red line. This means that our perceptron is able to classify points with a good accuracy.

What if reduce the number of training of our perceptron? You can try this by changing the value `500` by, let's say,`100`. What is the result? The perceptron does not do that well. This follow the intuition we elaborated when we first mentioned the training. More training a perceptron has, more accurate it will be (however, this is not always true with neural networks, as we will see later on).

EXERCISE: Reduce the number of time the perceptron is trained. Verify that varying the value 500 to lower value leads to some errors, illustrated at a mismatch between the red line and the area of colored points.

16 Measuring the precision

We have seen that the number of times we train a perceptron matters very much on how accurate the perceptron is able to classify points. How much training do we need to have a good precision? Keeping track of the precision and the training is essential to see how good our system is to do some classification.

Consider the script:

```
learningCurve := OrderedCollection new.
f := [ :x | (-2 * x) - 3 ].

0 to: 2000 by: 10 do: [ :nbOfTrained |
    r := Random new seed: 42.
    p := Neuron new.
```

```
    p weights: { 1 . 2 }.
    p bias: -1.

    nbOfTrained timesRepeat: [
        anX := (r nextInt: 50) - 25.
        anY := (r nextInt: 50) - 25.
        trainedOutput := (f value: anX) >= anY ifTrue: [ 1 ] ifFalse: [
            0 ].
        p train: (Array with: anX with: anY) desiredOutput:
            trainedOutput ].

    nbOfGood := 0.
    nbOfTries := 1000.
    nbOfTries timesRepeat: [
        anX := (r nextInt: 50) - 25.
        anY := (r nextInt: 50)- 25.
        realOutput := (f value: anX) >= anY ifTrue: [ 1 ] ifFalse: [ 0
            ].
        ((p feed: { anX . anY }) - realOutput) abs < 0.2
            ifTrue: [ nbOfGood := nbOfGood + 1 ].
    ].
    learningCurve add: { nbOfTrained . (nbOfGood / nbOfTries) }.
].

g := RTGrapher new.
d := RTData new.
d noDot.
d connectColor: Color blue.
d points: learningCurve.
d x: #first.
d y: #second.
g add: d.
g axisY title: 'Precision'.
g axisX noDecimal; title: 'Training iteration'.
g
```

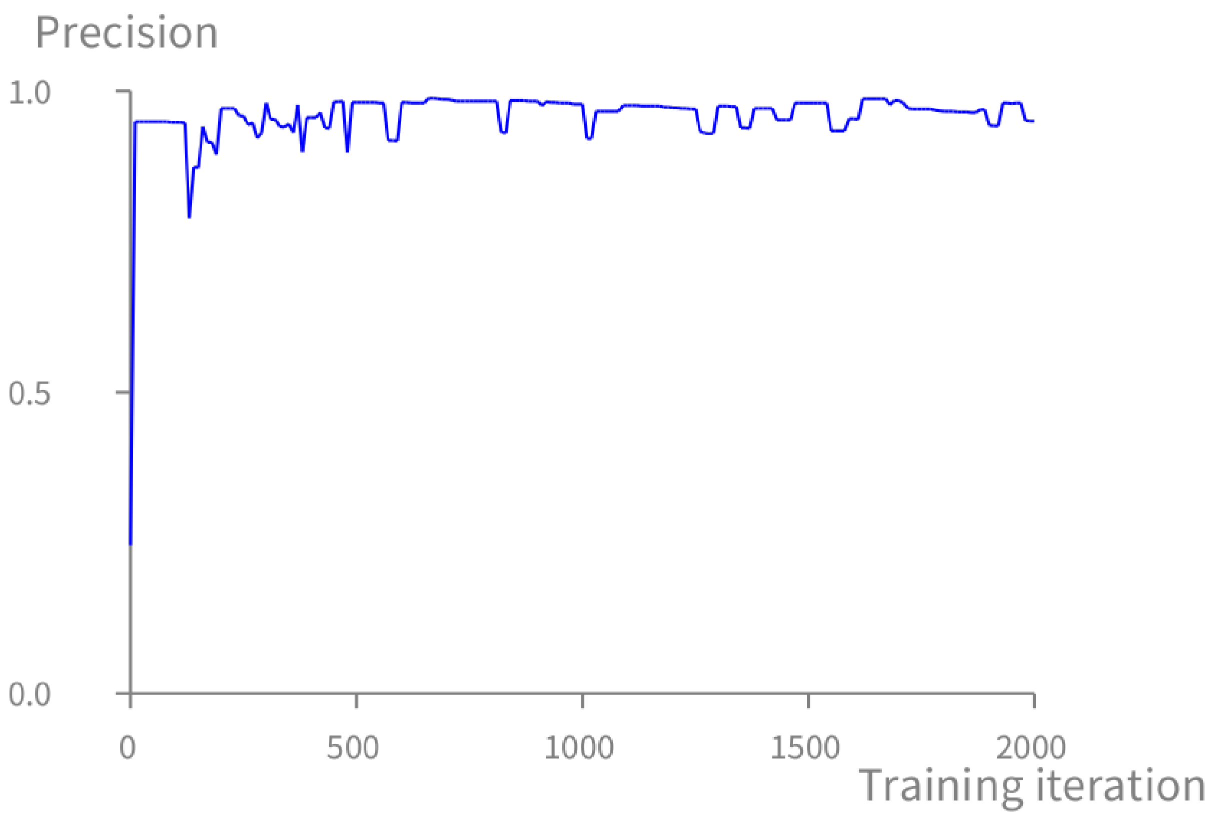

Figure 12: *perceptronPrecision*

The script produces a curve with the precision on the Y-axis and the number of trainings on the X-axis. We see that

17 What have we have

This chapter covers the following topics: - *Providing the concept of perceptron.* We have seen what is a perceptron. The perceptron is an essential abstraction on which we will built on top of in the next chapters. - *A step-by-step guide on programming with Pharo.* While we implemented the perceptron, we have sketched out how programming happens in Pharo. This chapter is by no means an introduction to Pharo. Instead, it is an overview on how to use the Pharo programming environment. In particular, we have seen how to write code using the system browser and how to run code using the playground. These two tools are fundamental and deserve to be well understood. - *Implemented a perceptron.* We implemented and tested the perceptron. Testing is important as it is a way to formalize the behavior we wish for the perceptron. - *Making a perceptron learn.* We have seen a rudimentary way to make a perceptron learn. It is rather simple, but, as we will see in the future chapters, the very same technique can bring us very far.

18 Exercises

- We have seen how the perceptron can be used to implement some logical gates. In particular, we have seen how the AND, OR, and NOT can be implemented. What about the XOR gate? Can you train a perceptron to learn the XOR behavior? (As a remainder, we have `0 XOR 0 = 0`, `0 XOR 1 = 1`, `1 XOR 0 = 1`, and `1 XOR 1 = 0`).
- We have seen how five perceptrons may be combined to form a digital comparator. Do you think you can train the combination of these five prceptrons as a whole to learn the behavior of the digital comparator?

19 Further reading about Pharo

Pharo is a wonderful programming language and programming environment. This first chapter may be used as an introduction to programming with Pharo. However, it is highly recommended to seek for complementary reading in order to feel confortable with Pharo. In particular, the *Pharo by example* book is an excellente introduction to learn and master Pharo. The website *http://books.pharo.org* contains a free copy of the book. Check it out!

Artificial Neuron

In the previous chapter we have seen how a perceptron operates and how a simple learning algorithm can be implemented. However, the perceptron, as we have seen it, has some serious limitations, which will motivate us to formulate a more robust artificial neuron, called the sigmoid neuron.

20 Limit of the Perceptron

A perceptron works well as an independent small machine. We have seen that we can compose a few perceptrons to express a complex behavior such as the digital comparator. We have also seen that a single perceptron can learn a behavior that is not too complex. However, there are two main restrictions with combining perceptrons:

- Perceptrons only output 0 or 1. In case we chain some perceptrons, using binary values significantly reduce the space we live in. Not everything in this work can be reduced as a set of 0 and 1.

- A chain of perceptrons cannot learn. In particular, a perceptron does not work for backpropagation, the most common learning algorithm for supervised learning using gradient descent.

We have written that $z = w.x + b$, for which w is a vector of weights, b a vector of bias, and x the input vector. We said that the output of perceptron is 1 if $z > 0$, else the output is 0. One important problem with the formulation of the perceptron is that a small variation of z can produce a large variation of the output: the output can goes from 0 to 1, or to 1 from 0.

Learning algorithms that are commonly employed in neural networks require a very important property: a small variation of z *must* produce a small variation of the output. And the perceptron does not fulfill this since a small variation of z can produce a large variation of the output.

21 Activation Function

One way to improve the learning ability of a perceptron is to structure the behavior of perceptron a bit. Let's introduce a function called σ. The perceptron behavior can therefore be summarized as: $\sigma(z) = 1$ if $z > 0$, else $\sigma(z) = 0$.

By adding the σ function, we are separating the computation of $w.x + b$ from the conditional. We call σ the *activation function*. It describes the activation of the perceptron (*i.e.,* when it fires 1) according to the value of z.

The activation function used by the perceptron is called the *step function*.

22 The Sigmoid Neuron

We described the perceptron as a powerful machine with a limitation to learn when combined with other perceptrons. We will therefore adopt another activation function. Consider the function $\sigma(z) = \frac{1}{1+e^{-z}}$.

This function can be plotted as:

[SigmoidNeuron]: 03-Neuron/figures/sigmoid.png "Image Title" {width = 200} [SigmoidNeuron][SigmoidNeuron]

This sigmoid function has several advantages:

- It is differentiable everywhere on the curve. Since its curve has no angle, we can easily draw a straight line for any value z that indicates the slope of $\sigma(z)$. When plotted, $\sigma(z)$ is very smooth by having no angle, which is a very good property.
- Its derivative has some interesting properties, as we will see later.
- It is an acceptable mathematical representation of a biological neuron behavior.

Another interesting aspect, is that the sigmoid function behave similarly than the step function for very small and very large z values. As a consequence, a small increment in z will produce a small variation of $\sigma(z)$.

The training has to be slightly adjusted to take advantage of the fact that $\sigma(z)$ is derivable.

23 Implementing the activation functions

In the previous chapter we have defined the class `Neuron`. We will improve this class to accept an activation function. First, let's introduce a small class hierarchy for activation functions.

Let's define the abstract class `ActivationFunction`:

```
Object subclass: #ActivationFunction
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'NeuralNetwork'
```

An activation function object has two main responsibility: computing (i) the activation value and (ii) the transfer derivative. This transfer derivative is essential for the backpropagation learning algorithm.

We define the following two abstract methods:

```
ActivationFunction>>eval: z
    ^ self subclassResponsibility
```

```
ActivationFunction>>derivative: z
    ^ self subclassResponsibility
```

We can now define the two activation functions. The sigmoid function can be defined as:

```
ActivationFunction subclass: #SigmoidAF
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'NeuralNetwork'
```

```
SigmoidAF>>>eval: z
    ^ 1 / (1 + z negated exp)
```

```
SigmoidAF>>>derivative: z
    | t |
    t := self eval: z.
```

```
    ^ t * (1 - t)
```

$\sigma(z)' = \sigma(z) * (1 - \sigma(z))$

The step function can be defined as:

```
ActivationFunction subclass: #StepAF
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'NeuralNetwork'
```

```
StepAF>>>eval: z
    ^ (z > 0) ifTrue: [ 1 ] ifFalse: [ 0 ]
```

```
StepAF>>>derivative: z
    ^ z
```

24 Extending the neuron with the activation functions

We can now extend our definition of neuron to use an activation function. We can do so by adding a new instance variable `activationFunction` to `Neuron`:

```
Object subclass: #Neuron
    instanceVariableNames: 'weights bias activationFunction'
    classVariableNames: ''
    package: 'NeuralNetwork'
```

Feeding has to be adapted:

```
Neuron>>feed: inputs
    | z |
    z := (inputs with: weights collect: [ :x :w | x * w ]) sum + bias.
    ^ activationFunction eval: z
```

Similarly, the training has to consider the derivative:

```
Neuron>>train: inputs desiredOutput: desiredOutput
    | learningRate theError output delta |
    output := self feed: inputs.
    learningRate := 0.1.

```

```
    theError := desiredOutput - output.
    delta := theError * (activationFunction derivative: output).

    inputs withIndexDo: [ :anInput :index |
        weights at: index put: ((weights at: index) + (learningRate *
            delta * anInput)) ].

    bias := bias + (learningRate * delta)
```

We now need to initialize a neuron as being a sigmoid:

```
Neuron>>initialize
    super initialize.
    self sigmoid
```

We can also define the two utility methods:

```
Neuron>>sigmoid
    activationFunction := SigmoidAF new
```

```
Neuron>>step
    activationFunction := StepAF new
```

25 Adapting the existing tests

If you run `PerceptronTest` you will see that several of the test fail. The reason is that a neuron is initialized with a sigmoid activation function. We therefore need to adapt each test method, by adding a call to `step`. For example, the method `testAND` has to be rewritten:

```
PerceptronTest>>testAND
    | p |
    p := Neuron new.
    p step. "<= new line"
    p weights: { 1 . 1 }.
    p bias: -1.5.

    self assert: (p feed: { 0 . 0 }) equals: 0.
    self assert: (p feed: { 0 . 1 }) equals: 0.
    self assert: (p feed: { 1 . 0 }) equals: 0.
    self assert: (p feed: { 1 . 1 }) equals: 1.
```

Adding the call to `step` make the neuron behaves as a perceptron. Omitting this line would instead use a sigmoid neuron, and the tests would fail since the output would not exactly be `0` or `1`.

EXERCISE: Adapt all the test methods of `PerceptronTest`.

26 Testing the sigmoid neuron

Since the behavior of a sigmoid neuron is very similar to a perceptron, we will reuse some of the tests. Define the class `NeuronTest`:

```
TestCase subclass: #NeuronTest
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'NeuralNetwork'
```

We can then train a neuron to learn some logical gates. The following method is very similar to what we have seen with the perceptron:

```
NeuronTest>>testTrainingAND
    | p |
    p := Neuron new.
    p weights: {-1 . -1}.
    p bias: 2.

    5000
        timesRepeat: [
            p train: {0 . 0} desiredOutput: 0.
            p train: {0 . 1} desiredOutput: 0.
            p train: {1 . 0} desiredOutput: 0.
            p train: {1 . 1} desiredOutput: 1 ].

    self assert: ((p feed: {0 . 0}) closeTo: 0 precision: 0.1).
    self assert: ((p feed: {0 . 1}) closeTo: 0 precision: 0.1).
    self assert: ((p feed: {1 . 0}) closeTo: 0 precision: 0.1).
    self assert: ((p feed: {1 . 1}) closeTo: 1 precision: 0.1).
```

There are two differences: - The number of epochs is significantly increased. The reason is that the sigmoid neuron learns slower than the perceptron. - The result of feeding the neuron is compared using the call `closeTo:precision:`.

Similarly we can train a sigmoid neuron to learn the OR behavior:

```
NeuronTest>>testTrainingOR
    | p |
    p := Neuron new.
    p weights: {-1 . -1}.
    p bias: 2.

    5000
        timesRepeat: [
            p train: {0 . 0} desiredOutput: 0.
            p train: {0 . 1} desiredOutput: 1.
            p train: {1 . 0} desiredOutput: 1.
            p train: {1 . 1} desiredOutput: 1 ].

    self assert: ((p feed: {0 . 0}) closeTo: 0 precision: 0.1).
    self assert: ((p feed: {0 . 1}) closeTo: 1 precision: 0.1).
    self assert: ((p feed: {1 . 0}) closeTo: 1 precision: 0.1).
    self assert: ((p feed: {1 . 1}) closeTo: 1 precision: 0.1).
```

27 Sigmoid neuron is slower to learn

This chapter is based on some limitation of the perceptron to be composed with other perceptrons. This has motivated us to formulate the sigmoid neuron. We see one drawback of the sigmoid neuron: it is slower to learn than the perceptron. We are here making a bet, which is trading efficiency for flexibility.

We can easily make the comparison between the sigmoid neuron and perceptron. Consider the following script:

```
learningCurve := OrderedCollection new.
0 to: 1000 do: [ :nbOfTrained |
    r := Random new seed: 42.
    p := Neuron new.
    p weights: {-1 . -1}.
    p bias: 2.

    nbOfTrained timesRepeat: [
        p train: {0 . 0} desiredOutput: 0.
        p train: {0 . 1} desiredOutput: 0.
        p train: {1 . 0} desiredOutput: 0.
        p train: {1 . 1} desiredOutput: 1 ].
```

```

    res :=  ((p feed: {0 . 0}) - 0) abs +
            ((p feed: {0 . 1}) - 0) abs +
            ((p feed: {1 . 0}) - 0) abs +
            ((p feed: {1 . 1}) - 1) abs.
     learningCurve add: res / 4.

].

g := RTGrapher new.
d := RTData new.
d noDot.
d connectColor: Color blue.
d points: learningCurve.
d y: #yourself.
g add: d.
g axisY title: 'Error'.
g axisX noDecimal; title: 'Epoch'.
g
```

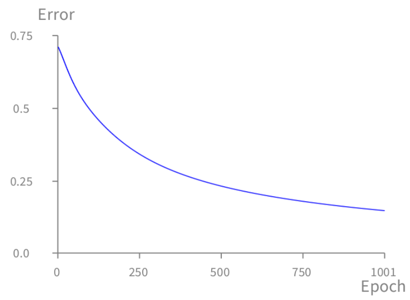

Figure 13: *SigmoidLearningAND*

Adding a line `p step` after the creation of the neuron will produce the error curve for the perceptron.

28 The derivative of the sigmoid function

We have $\sigma(x) = \frac{1}{1+e^{-x}}$. So, we also have:

$\frac{d}{dx}\sigma(x) = \frac{d}{dx}\frac{1}{1+e^{-x}}$

$= \frac{d}{dx}(1+e^{-z})^{-1}$

Since the derivative of x^n is nx^{n-1}, we have

$= -(1 + e^{-x})^{-2}(-e^{-x})$

By simplifying we have:

$= \frac{e^{-x}}{(1+e^{-x})^2}$

$= \frac{1}{1+e^{-x}} . \frac{e^{-x}}{1+e^{-x}}$

$= \frac{1}{1+e^{-x}} . \frac{(1+e^{-x})-1}{1+e^{-x}}$

$= \frac{1}{1+e^{-x}} . (\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}})$

$= \frac{1}{1+e^{-x}} . (1 - \frac{1}{1+e^{-x}})$

$= \sigma(x).(1 - \sigma(x))$

29 What have we have

This chapter covers the following topics:

- *Briefly discussed the limitation of the perceptron.* The perceptron cannot learn when combined with other perceptrons. Although we have not discussed this aspect further, you need to trust me for now. In the next chapter we will develop this further.
- *Definition of the sigmoid neuron.* The sigmoid neuron is an improvement of the perceptron since it can be combined with other sigmoid neurons and this combination can learn. In the next chapter we will detail the backpropagation algorithm, a central aspect when making a neural network learn.
- *Activation functions.* We have seen two activation functions, the step and sigmoid functions. Many other activation functions are around. We will develop activation functions later on in the book.

Neural Networks

In the previous chapter we have seen how to define single artificial neuron. This chapter belongs to the core of this book as it presents how to connect and give a meaning to a group of artificial neurons.

30 General architecture

An artificial neural network is a computing system inspired by biological neural networks which define animal brains. An artificial neural network is a collection of connected artificial neurons. Each connection between artificial neurons can transmit a signal from one to another. The artificial neuron that receives the signal can process it, and then signal neurons connected to it.

Artificial neural networks are commonly employed to perform particular tasks, including clustering, classification, prediction, and pattern recognition.

A neural network acquires knowledge through learning.

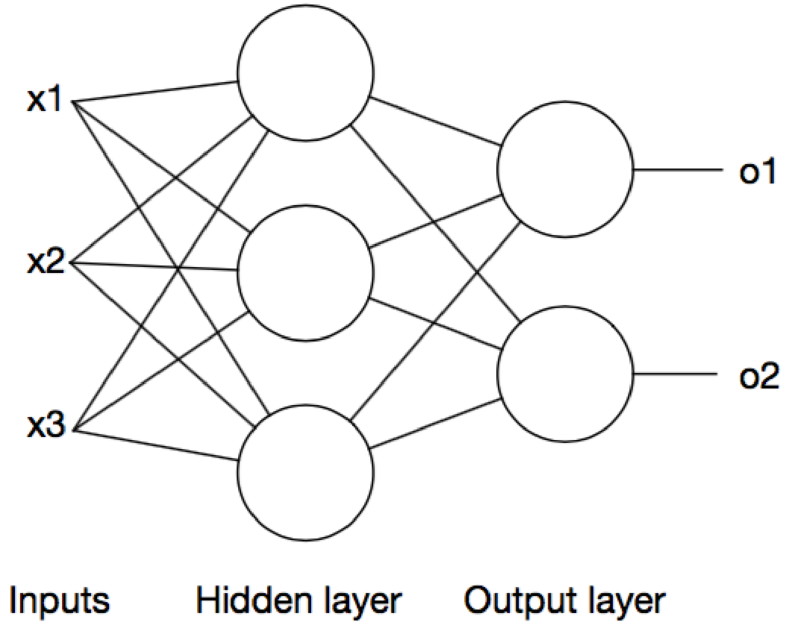

Figure 14: *generalStructure*

The figure above shows a network made of five neurons, three inputs, and two outputs. The left-most column is called the inputs. The networks contains three inputs, `x1`, `x2`, and `x3`. the middle of the network contains the hidden layers. The network above contains only one hidden layer, made of three neurons. The right-most part of the network is called output layer, and is made of two neurons.

We therefore needs three different components to model networks: - *Hidden layer* representing layers within - *Output layer* representing the right-most column. This is that column that spits out computed values - *Neural Network* made of several hidden layers and one output layer.

31 Neural layer

We define a layer as a set of neurons. Each layer knows about the preceding layer using the variable `previousLayer` and the following layer using `nextLayer`. The `learningRate` variable refers to the learning rate of the layer. We define the class `NeuronLayer` as follows:

```
Object subclass: #NeuronLayer
```

```
    instanceVariableNames: 'previousLayer nextLayer neurons
        learningRate'
    classVariableNames: ''
    package: 'NeuralNetwork'
```

We can set the learning rate of a layer as 0.5 per default. A neuron layer may be initialized using the following method:

```
NeuronLayer>>initializeNbOfNeurons: nbOfNeurons nbOfWeights:
    nbOfWeights using: random
    "Main method to initialize a neuron layer
    nbOfNeurons : number of neurons the layer should be made of
    nbOfWeights : number of weights each neuron should have
    random : a random number generator
    "
    | weights |
    learningRate := 0.5.
    neurons := (1 to: nbOfNeurons) collect: [ :i |
        weights := (1 to: nbOfWeights) collect: [ :ii | random next * 2
            - 1 ].
        Neuron new sigmoid; weights: weights; bias: (random next * 2 -
            1) ]
```

The method `initializeNbOfNeurons:nbOfWeights:using:` accepts three arguments. The first one, `nbOfNeurons` is an integer number representing the number of neurons the layer should contains. The second argument, `nbOfWeights`, is an integer that indicates the number of weights each neuron should have. This number of weights reflects the number of input values the layer is accepting. The last argument, `random`, is a random number generator.

The method set the learning rate to `0.5`, and create `nbOfNeurons` neurons, each having `nbOfWeights` weight value. Each weight is a random number between -1 and +1. Each neuron has a sigmoid activation function.

Forward feeding the layer is an essential operation. It consists in feeding each neurons and forwarding the values to the next value. We define the method `feed:` as:

```
NeuronLayer>>feed: someInputValues
    "Feed the neuron layer with some inputs"

    | someOutputs |
    someOutputs := neurons collect: [ :n | n feed: someInputValues ] as
        : Array.
    ^ self isOutputLayer
```

```
        ifTrue: [ someOutputs ]
        ifFalse: [ nextLayer feed: someOutputs ]
```

The method invokes `feed:` on each of its neurons. The results is then kept as an Array. The method then check if the layer is an output layer. If this is the case, the result of the method is simply the results of each neurons. If the layer is not an output (*i.e.,* it is an hidden layer), we forward feed the next layer.

We now need the following utility method:

```
NeuronLayer>>isOutputLayer
    "Return true of the layer is the output layer (i.e., the last layer
        in the network)"
    ^ self nextLayer isNil
```

We can now prepare a testbed for our new neuron layer class:

```
TestCase subclass: #NeuronLayerTest
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'NeuralNetwork'
```

A simple test may be:

```
NeuronLayerTest>>testBasic
    | nl result |
    nl := NeuronLayer new.
    nl initializeNbOfNeurons: 3 nbOfWeights: 4 using: (Random seed: 42)
        .

    self assert: nl isOutputLayer.

    result := nl feed: { 1 . 2 . 3 . 4 }.
    self assert: result size equals: 3.
    self assert: result closeTo: #(0 1.1277714465408457
        1.9863987178478386).
```

32 Neural network

We will represent a neural network as an instance of the class `NNetwork`:

```
Object subclass: #NNetwork
```

```
    instanceVariableNames: 'layers errors'
    classVariableNames: ''
    package: 'NeuralNetwork'
```

We define a neural network simply as a container of layers. We also add an `errors` instance variable that will be useful to trace the error during a learning phase.

The initialization of a network is done through the method `initialize`:

```
NNetwork>>initialize
    super initialize.
    layers := OrderedCollection new.
    errors := OrderedCollection new
```

Both the `layers` and `errors` instance variables are initialized with an empty collection. The variable `layers` will contains instance of the class `NeuronLayer` and `errors` will contains numerical values, representing the errors during the training process.

Adding a layer is simply done through the method `addLayer:`, which takes a layer as argument:

```
NNetwork>>addLayer: aNeuronLayer
    "Add a neural layer. The added layer is linked to the already added
        layers."
    layers ifNotEmpty: [
        aNeuronLayer previousLayer: layers last.
        layers last nextLayer: aNeuronLayer ].
    layers add: aNeuronLayer.
```

Layers are linked to each other. When a layer is added, it is linked to the previous layer and that layer is linked to the added layer.

```
NNetwork>>feed: someInputValues
    "Feed the first layer with the provided inputs"
    ^ self firstLayer feed: someInputValues
```

We add the necessary to easily create a neural network. In case we wish to build a network with one hidden layer and one output layer:

```
NNetwork>>configure: nbOfInputs hidden: nbOfNeurons nbOfOutputs:
   nbOfOutput
    "Configure the network with the given parameters
    The network has only one hidden layer"
    | random |
    random := Random seed: 1.
```

```
    self addLayer: (NeuronLayer new initializeNbOfNeurons: nbOfNeurons nbOfWeights: nbOfInputs using: random).
    self addLayer: (NeuronLayer new initializeNbOfNeurons: nbOfOutput nbOfWeights: nbOfNeurons using: random).
```

In case we wish to have two hidden layers and one output layer:

```
NNetwork>>configure: nbOfInputs hidden: nbOfNeurons1 hidden: nbOfNeurons2 nbOfOutputs: nbOfOutput
    "Configure the network with the given parameters
    The network has only one hidden layer"
    | random |
    random := Random seed: 42.
    self addLayer: (NeuronLayer new initializeNbOfNeurons: nbOfNeurons1 nbOfWeights: nbOfInputs using: random).
    self addLayer: (NeuronLayer new initializeNbOfNeurons: nbOfNeurons2 nbOfWeights: nbOfNeurons1 using: random).
    self addLayer: (NeuronLayer new initializeNbOfNeurons: nbOfOutput nbOfWeights: nbOfNeurons2 using: random).
```

We can now tests our network implementation:

```
TestCase subclass: #NNetworkTest
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'NeuralNetwork'
```

```
NNetworkTest>>testBasic
    | n |
    n := NNetwork new.
    n configure: 2 hidden: 2 nbOfOutputs: 1.
    self assert: (n feed: { 1 . 3 }) closeTo: { 0.3585928116732244З }
```

As you can see, `testBasic` is rather simplistic. It builds a simple network with two inputs, one hidden layer made of 2 neurons, and an output layer with only one neuron, and run the forward feeding.

33 Backpropagation

Backpropagation is an algorithm commonly employed to train neural network. In this section we will focus on implementing the algorithm. The training process is composed of three steps: 1.Forward feeding the inputs 1.Backward propagating the errors through the network 1.Updating the neurons weights and biases

The first phase is mostly implemented by the method `NNetwork>>feed:`, however, we need to improve our neuron to keep the output. During the forward feeding (*i.e.,* when the method `feed:` is called), an output is produced by each neuron. This output has to be compared with an expected output. Making the network learn is based on the difference between the actual output of a neuron and the expected output. Each neuron has therefore to keep a reference to the output.

We add two variables, `delta` and `output`, in the class `Neuron`,

```
Object subclass: #Neuron
    instanceVariableNames: 'weights bias delta output
        activationFunction'
    classVariableNames: ''
    package: 'NeuralNetwork'
```

We now rewrite the method `feed:` on the class `Neuron`:

```
Neuron>>feed: inputs
    | z |
    z := (inputs with: weights collect: [ :x :w | x * w ]) sum + bias.
    output := activationFunction eval: z.
    ^ output
```

At that stage, it is important to run the unit tests we have run. We are now done with the first phase of the backpropagation.

The second phase consists in propagating the errors computed at the output layer back in the network. We define the following method:

```
NNetwork>>backwardPropagateError: expected
    self outputLayer backwardPropagateError: expected
```

And the following helper method:

```
NNetwork>>outputLayer
    "Return the output layer, which is also the last layer"
    ^ layers last
```

```
NeuronLayer>>backwardPropagateError: expected
    "This is a recursive method. The back propagation begins with the
        output layer (i.e., the last later)"
    "We are in the output layer"
```

```
    neurons with: expected do: [ :neuron :exp |
        | theError |
        theError := exp - neuron output.
        neuron adjustDeltaWith: theError ].

    "We iterate"
    self previousLayer notNil
        ifTrue: [
            self previousLayer backwardPropagateError ].
```

```
NeuronLayer>>backwardPropagateError
    "This is a recursive method. The back propagation begins with the
       output layer (i.e., the last later)"

    "We are in an hidden layer"
    neurons doWithIndex: [ :neuron :j |
        | theError |
        theError := 0.0.
        self nextLayer neurons do: [ :nextNeuron |
            theError := theError + ((nextNeuron weights at: j) *
               nextNeuron delta)
        ].
        neuron adjustDeltaWith: theError
    ].

    self previousLayer notNil
        ifTrue: [
            self previousLayer backwardPropagateError ].
```

We also need the following helper method:

```
Neuron>>adjustDeltaWith: anError
    delta := anError * (activationFunction derivative: output)
```

The recursion happens in this method:

```
NeuronLayer>>backwardPropagateError
    "This is a recursive method. The back propagation begins with the
       output layer"

    "We are in an hidden layer"
    neurons doWithIndex: [ :neuron :j |
        | theError |
```

```
        theError := 0.0.
        self nextLayer neurons do: [ :nextNeuron |
            theError := theError + ((nextNeuron weights at: j) * nextNeuron delta)
        ].
        neuron adjustDeltaWith: theError
    ].

    previousLayer notNil
        ifTrue: [
             previousLayer backwardPropagateError ].
```

The third phase is rather simple:

```
NNetwork>>updateWeight: initialInputs
    "Update the weights of the neurons using the initial inputs"
    layers first updateWeight: initialInputs
```

```
NeuronLayer>>updateWeight: initialInputs
    "Update the weights of the neuron based on the set of initial input. This method assumes that the receiver of the message invoking that method is the first hidden layer."
    | inputs |
    inputs := initialInputs.

    neurons do: [ :n |
        n adjustWeightWithInput: inputs learningRate: learningRate.
        n adjustBiasUsingLearningRate: learningRate ].

    self nextLayer ifNotNil: [
        self nextLayer updateWeight ]
```

And we need the following methods to update a neuron state:

```
Neuron>>adjustWeightWithInput: inputs learningRate: learningRate
    inputs withIndexDo: [ :anInput :index |
        weights at: index put: ((weights at: index) + (learningRate * delta * anInput)) ]
```

```
Neuron>>adjustBiasUsingLearningRate: learningRate
    bias := bias + (learningRate * delta)
```

We are now ready to hook the backpropagation phases all together:

```
NNetwork>>train: someInputs desiredOutputs: desiredOutputs
    "Train the neural network with a set of inputs and some expected
       output"
    | realOutputs t |
    realOutputs := self feed: someInputs.
    t := (1 to: desiredOutputs size) collect:
            [ :i | ((desiredOutputs at: i) - (realOutputs at: i))
               raisedTo: 2 ].
    self backwardPropagateError: desiredOutputs.
    self updateWeight: someInputs.
```

We can now test our network with the XOR example:

```
NNetworkTest>>testXOR
    | n |
    n := NNetwork new.
    n configure: 2 hidden: 3 nbOfOutputs: 1.

    10000 timesRepeat: [
        n train: { 0 . 0 } desiredOutputs: { 0 }.
        n train: { 0 . 1 } desiredOutputs: { 1 }.
        n train: { 1 . 0 } desiredOutputs: { 1 }.
        n train: { 1 . 1 } desiredOutputs: { 0 }.
    ].

    self assert: ((n feed: { 1 . 1 }) first - 0.025) < 0.01.
    self assert: ((n feed: { 0 . 1 }) first - 1) < 0.01.
    self assert: ((n feed: { 1 . 0 }) first - 1) < 0.01.
    self assert: ((n feed: { 0 . 0 }) first - 0.025) < 0.01.
```

If you try to decrease the `10000` to a low value, `10` for example, the network does not receive enough training and the test ultimately fails.

34 Cost function

A cost function is a measure of how well a neural networks leans. A cost function is a single value since it rates how good the neural network did as a whole.

Genetic Algorithm

Evolution of life on Earth is one of the most spectacular evolutionary system that we know. The way life has evolved on Earth has triggered numerous attempts to identify necessary ingredients for a successful evolution of complex systems. Numerous algorithms have been produced as a result to capture evolution mechanisms.

35 Evolutionary algorithms

"'Evolutionary algorithms"' uses mechanisms inspired by biological evolution, which typically include reproduction, mutation, recombination, and selection. The flow chart of an evolutionary algorithm is relatively simple (evolutionaryAlgorithm). A population of individual is first produced. Subsequently, a selection is carried out to select relevant individual from the initial population. From these selected individual, that we call "parents", offspring are generated to finally replace the initial population. Assuming an adequate configuration, a succession of generations may lead to a convergence to a set of relevant individuals.

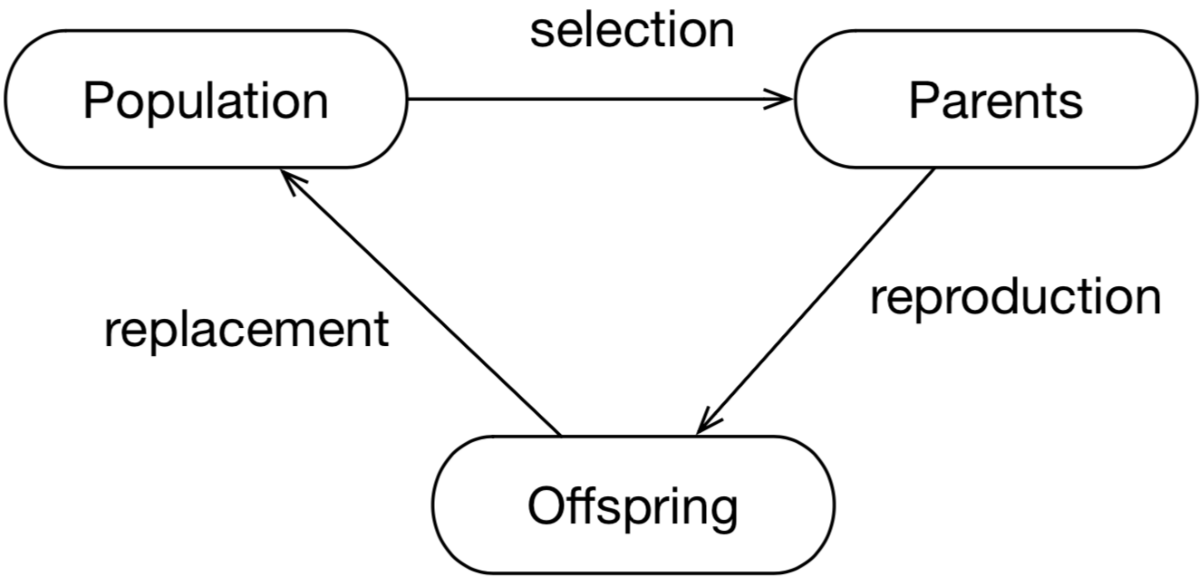

Figure 15: *evolutionaryAlgorithm*

Several evolutionary algorithms have been proposed: - *Ant colony optimization*: technique to find a solution to problems that can be reduced to finding good path trough a graph. - *Genetic algorithm*: Technique to optimize and search for a solution in a large space by relying on the mutation and crossover genetic operations. - *Genetic programming*: Similarly to genetic algorithm, genetic programming is about finding an executable program that achieve a good fitness value for a given objective function. - *Grammatical evolution*: This technique is similar to genetic programming, expect it has a particular set of genetic operators.

Evolutionary algorithms have numerous advantages: - An evolutionary algorithm is not tied to a particular problem to solve. The exact same algorithm may be applied to solve very different problems. - An algorithm may be combined with other optimization techniques. A classical example is to use an evolutionary algorithm to find weights of a neural networks. - An evolutionary algorithm can solve some problem for which no solution can be easily determined by a human expert. In particular, it may solve a problem without specificying the form or structure of the solution in advance.

36 Genetic Algorithm

A genetic algorithm is a simulation of evolution that uses biological genetic operations. The notion of genetic algorithm appeared in 1967, in J. D. Bagley's thesis "The Behavior of Adaptive Systems Which Employ Genetic and Correlative Algorithms".

A genetic algorithm is often considered as an optimization method, *i.e.,* assuming a function $f : X \to R$, X being a multi-dimensional space, the genetic algorithm finds x such as $f(x)$ is maximal.

37 More to read

- "Genetic Systems Programming: Theory and Experiences", 2006, edited by Ajith Abraham

Example of GA applied to the traveling salesman problem:

http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5

38 Difference with other techniques

For example, Simulated Annealing (http://www.theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6)