1 Introduction

1.1 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.*

1.2 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

1.3 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.

1.4 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.

1.5 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,

1.6 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.

1.7 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

2 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.

2.1 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).

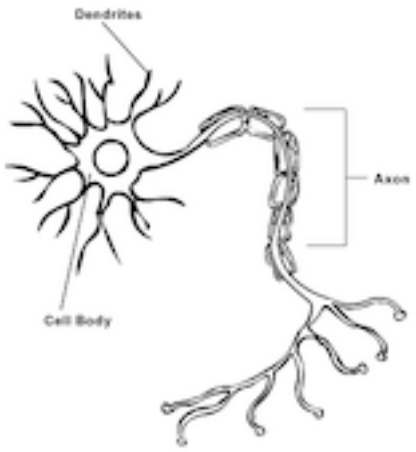

Figure 2.1: *Neuron*

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.

2.2 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).

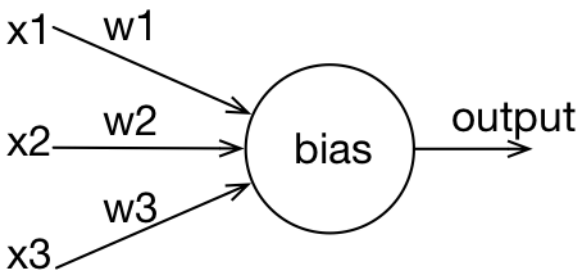

Figure 2.2: *Perceptron*

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 = x^2 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

2.3 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.

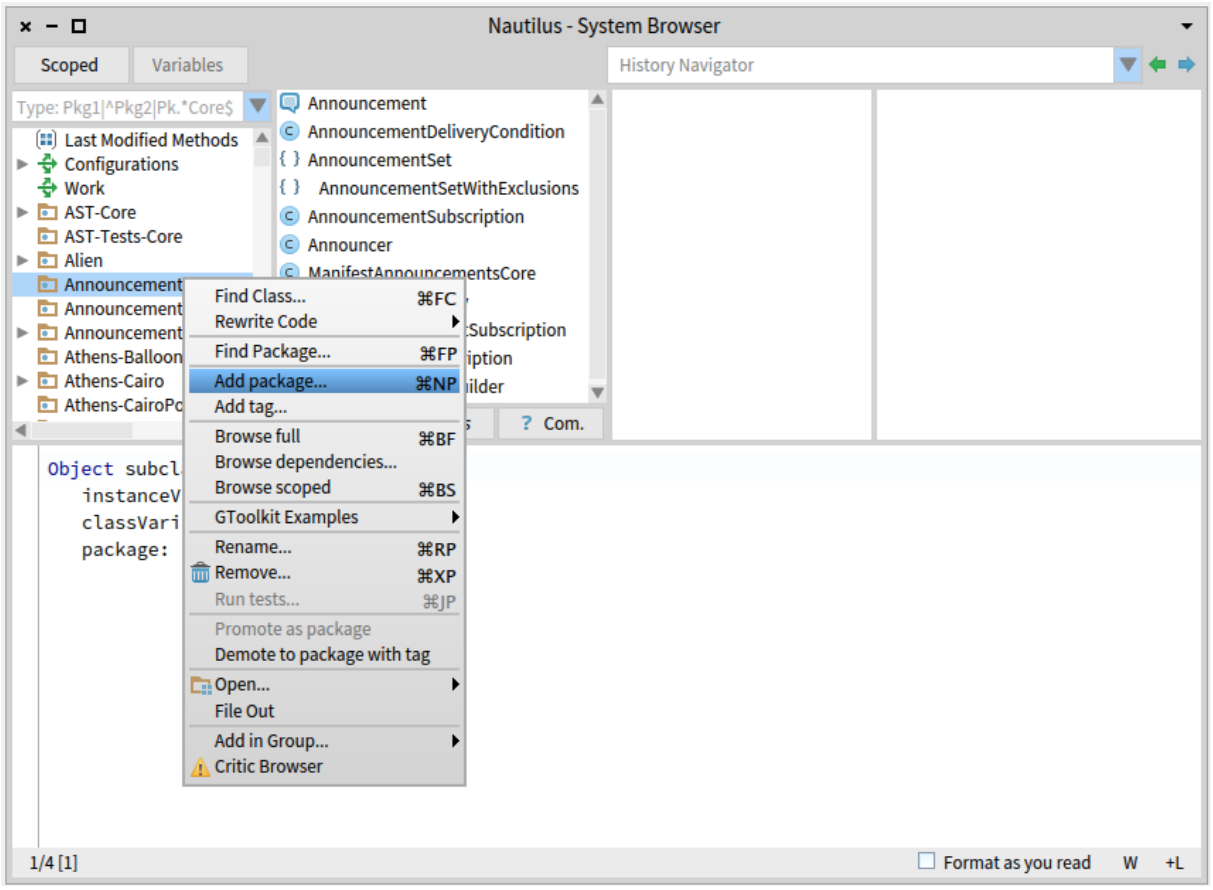

Figure 2.3: *systemBrowser*

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

```
1 Object subclass: #Perceptron
2     instanceVariableNames: 'weights bias'
3     classVariableNames: ''
4     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.

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

```
1 Perceptron>>weights: someWeightsAsNumbers
2     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* `Perceptron>>`. 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.

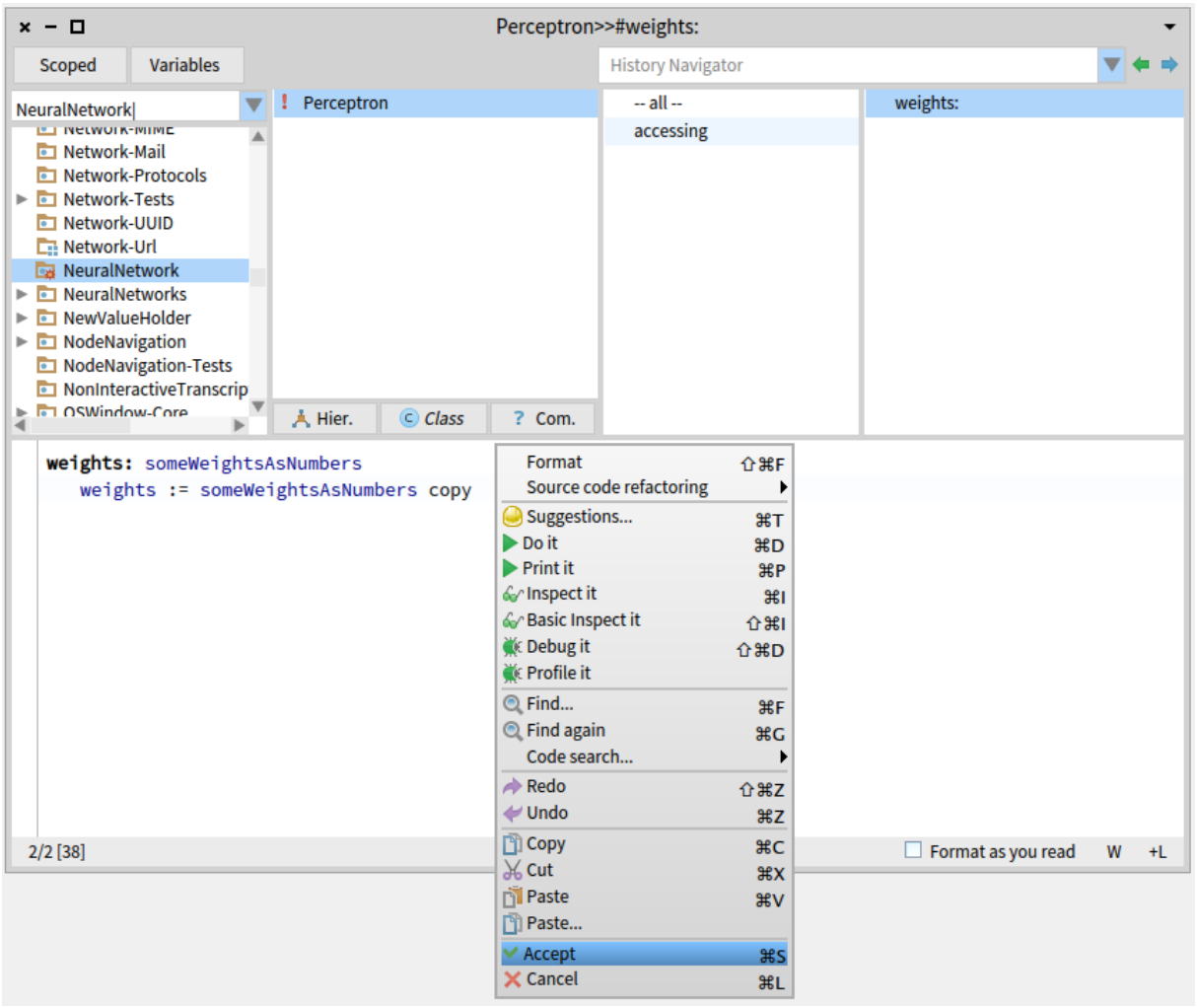

Figure 2.4: *systemBrowserAndMethodWeight*

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

```
1 Perceptron>>weights
2     ^ 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:

```
1 Perceptron>>bias: aNumber
2     bias := aNumber
```

And the reading may be defined using:

```
Perceptron>>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:

```
Perceptron>>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).

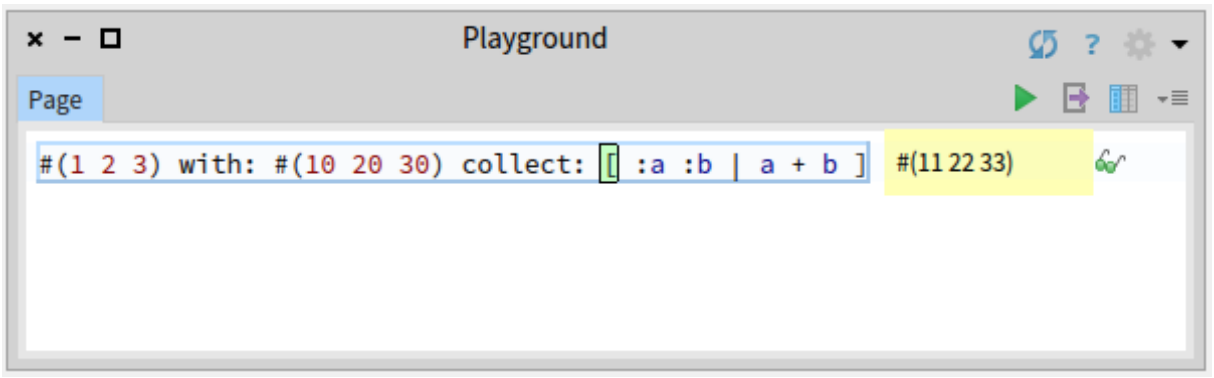

Figure 2.5: *playground*

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

```
p := Perceptron new.
p weights: #(1 2).
```

```
3 p bias: -2.
4 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`).

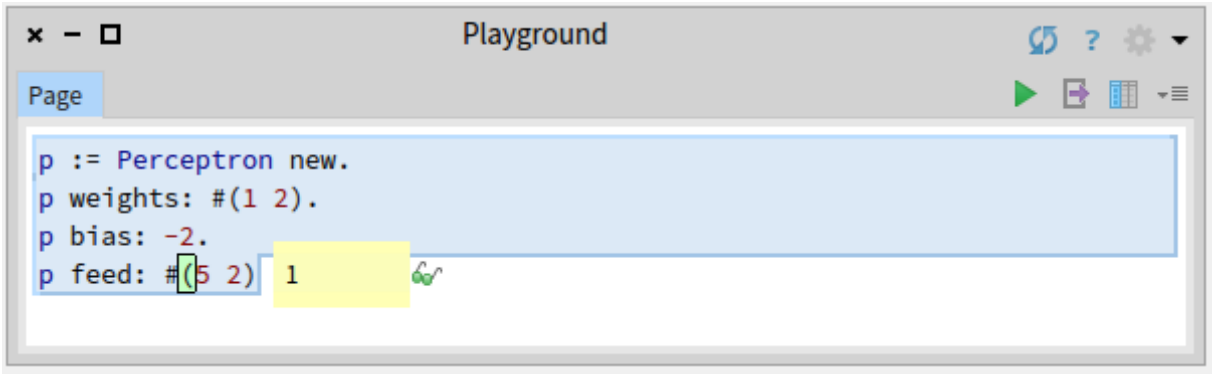

Figure 2.6: *playgroundAndPerceptron*

2.4 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:

```
1 TestCase subclass: #PerceptronTest
2     instanceVariableNames: ''
3     classVariableNames: ''
4     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:

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

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

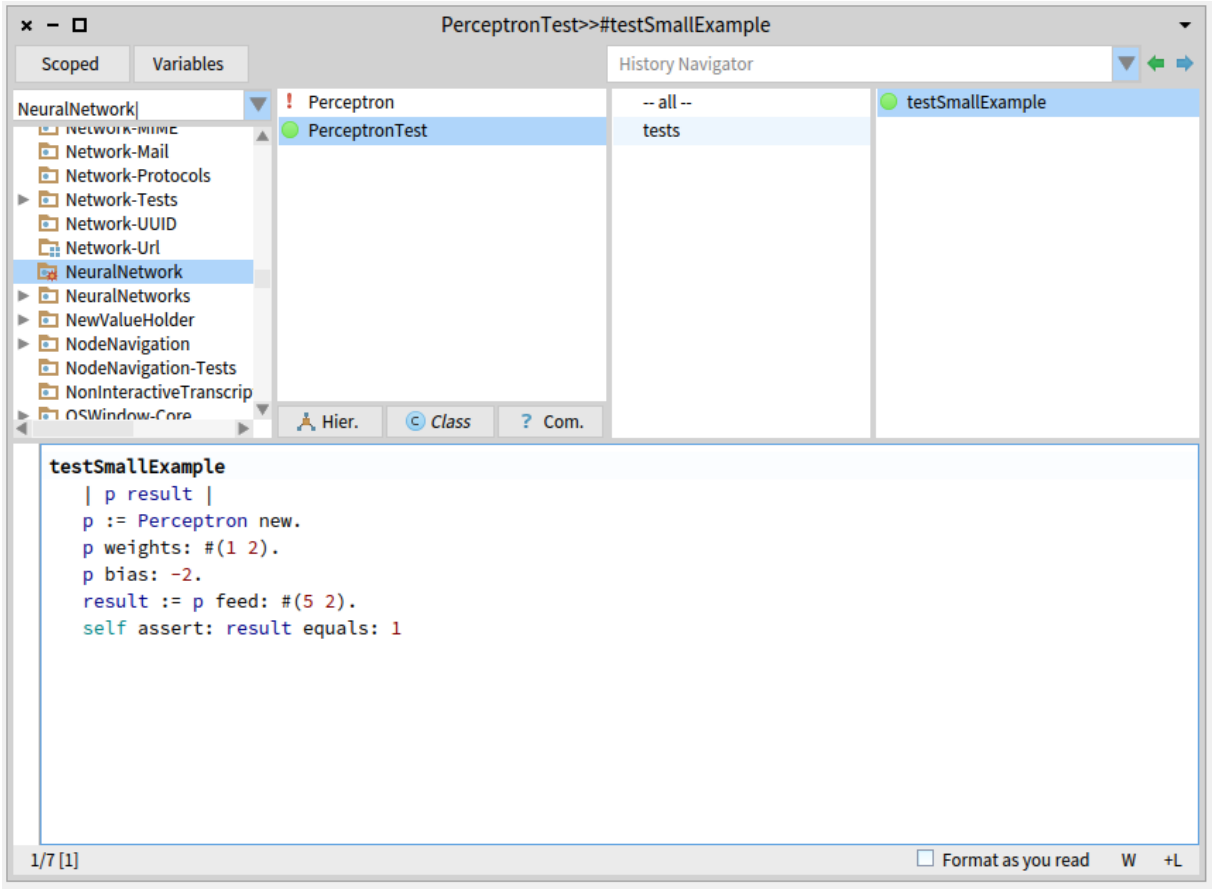

Figure 2.7: *testingPerceptron01*

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`.

2.5 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:

```
1  PerceptronTest>>testAND
2      | p |
3      p := Perceptron new.
4      p weights: { 1 . 1 }.
5      p bias: -1.5.
6
7      self assert: (p feed: { 0 . 0 }) equals: 0.
8      self assert: (p feed: { 0 . 1 }) equals: 0.
9      self assert: (p feed: { 1 . 0 }) equals: 0.
10     self assert: (p feed: { 1 . 1 }) equals: 1.
```

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

```
1  PerceptronTest>>testOR
2      | p |
3      p := Perceptron new.
4      p weights: { 1 . 1 }.
5      p bias: -0.5.
6
7      self assert: (p feed: { 0 . 0 }) equals: 0.
8      self assert: (p feed: { 0 . 1 }) equals: 1.
9      self assert: (p feed: { 1 . 0 }) equals: 1.
10     self assert: (p feed: { 1 . 1 }) equals: 1.
```

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

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

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

2.6 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

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

The circuit is defined as:

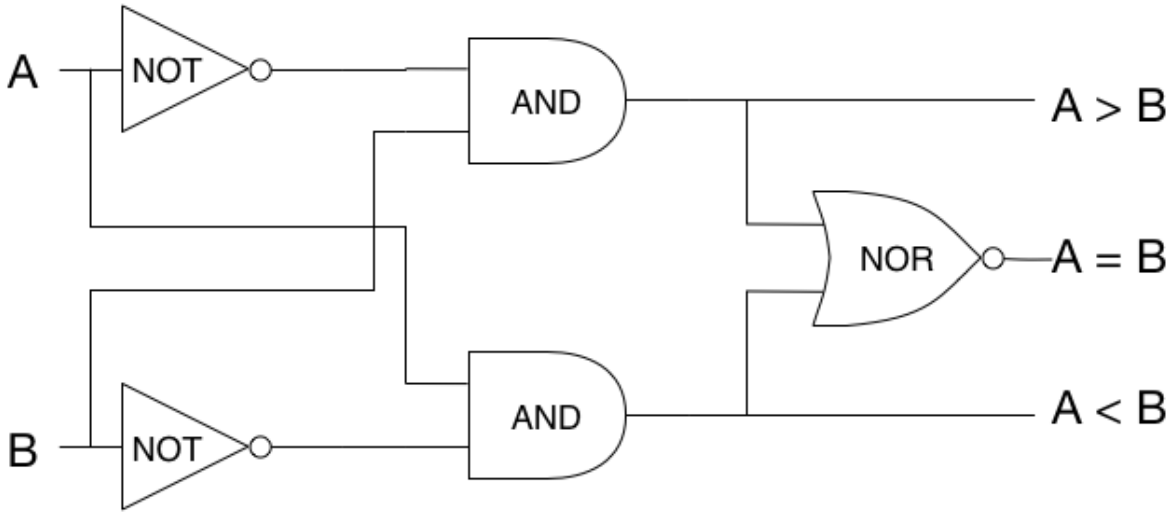

Figure 2.8: *digitalComparator*

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:

```
1  PerceptronTest>>digitalComparator: inputs
2      "Return an array of three elements"
3      | not and nor A B AgB AeB AlB notA notB |
4      A := inputs first.
5      B := inputs second.
6
7      and := Perceptron new weights: { 1 . 1 }; bias: -1.5.
8      not := Perceptron new weights: { -1 }; bias: 0.5.
9      nor := Perceptron new weights: { -1 . -1 }; bias: 0.5.
10
11     notA := not feed: { A }.
12     notB := not feed: { B }.
13
14     AgB := and feed: { notA . B }.
```

```
15      AlB := and feed: { A . notB }.
16      AeB := nor feed: { AgB . AlB }.
17      ^ { 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:

```
1  PerceptronTest>>testDigitalComparador
2      self assert: (self digitalComparator: { 0 . 0 }) equals: { 0 . 1 .
           0 }.
3      self assert: (self digitalComparator: { 0 . 1 }) equals: { 1 . 0 .
           0 }.
4      self assert: (self digitalComparator: { 1 . 0 }) equals: { 0 . 0 .
           1 }.
5      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.

2.7 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:

```
Perceptron>>train: inputs desiredOutput: desiredOutput
    | realOutput difference oldWeight currentInput learningRate
       newWeight |
    realOutput := self feed: inputs.
    difference := desiredOutput - realOutput.
    learningRate := 0.1.
    1 to: weights size do: [ :index |
        oldWeight := weights at: index.
        currentInput := inputs at: index.
        newWeight := oldWeight + (learningRate * currentInput *
           difference).
        weights at: index put: newWeight.

        bias := bias + (learningRate * difference). ]
```

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 := MPPerceptron 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 := Perceptron 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 := Perceptron 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!

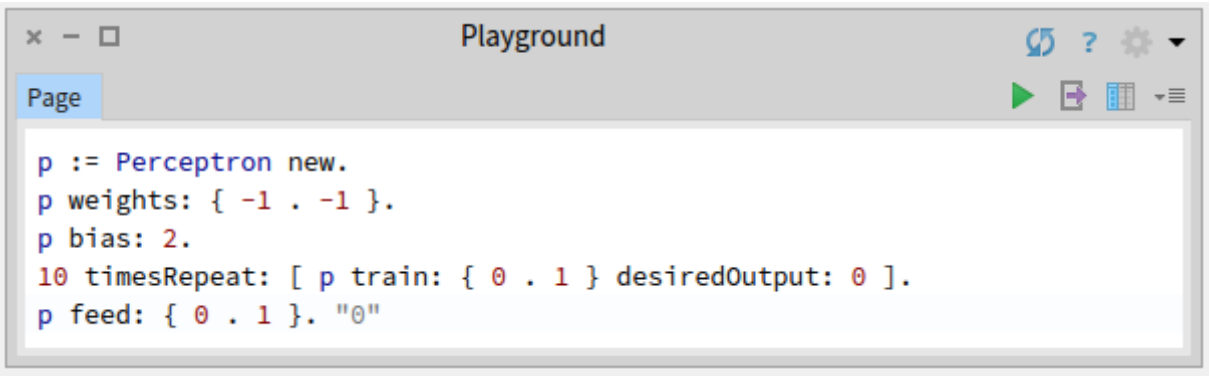

Figure 2.9: *playgroundWithLearningPerceptron*

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

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

    25 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 25 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 25 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?

2.8 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.

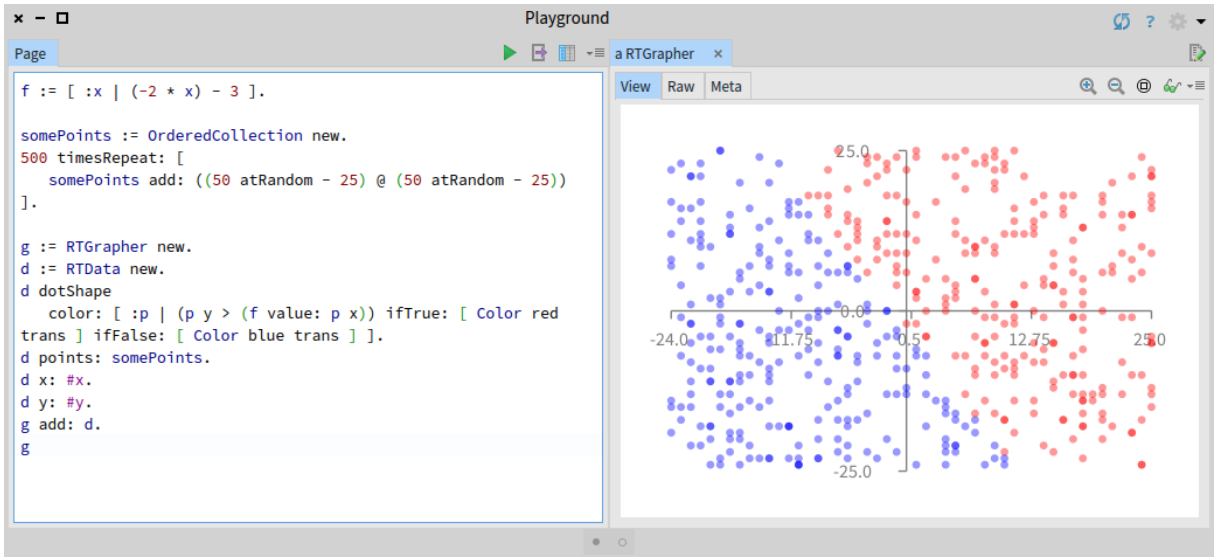

Figure 2.10: *simpleLine*

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
```

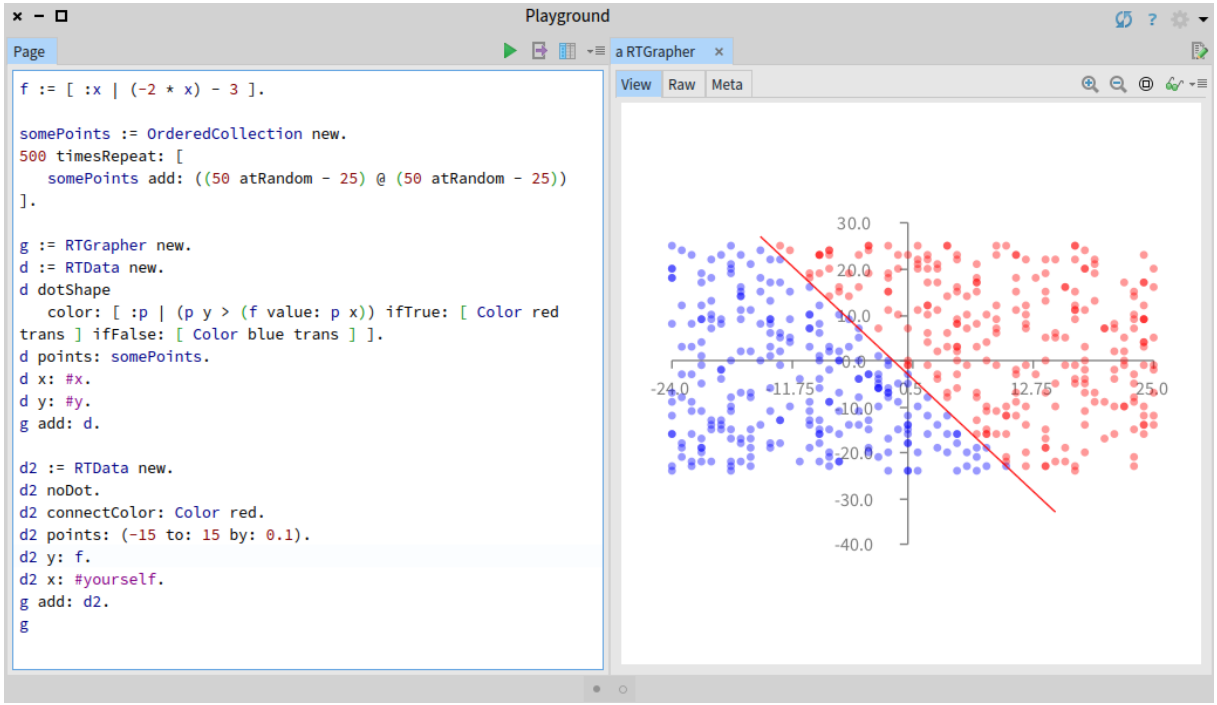

Figure 2.11: *simpleLine2*

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:

```
1  f := [ :x | (-2 * x) - 3 ].
2  p := Perceptron new.
3  p weights: { 1 . 2 }.
4  p bias: -1.
5  r := Random new seed: 42.
6
7  "We are training the perceptron"
8  500 timesRepeat: [
9      anX := (r nextInt: 50) - 25.
10     anY := (r nextInt: 50) - 25.
11     designedOutput := (f value: anX) >= anY
12                                 ifTrue: [ 1 ] ifFalse: [ 0 ].
13     p train: { anX . anY } desiredOutput: designedOutput
14 ].
15
16 "Test points"
17 testPoints := OrderedCollection new.
18 2000 timesRepeat: [
19     testPoints add: { ((r nextInt: 50) - 25) . ((r nextInt: 50) - 25) }
```

```
20 ].
21
22 g := RTGrapher new.
23 d := RTData new.
24 d dotShape
25     color: [ :point | (p feed: point) > 0.5
26                     ifTrue: [ Color red trans ]
27                     ifFalse: [ Color blue trans ] ].
28 d points: testPoints.
29 d x: #first.
30 d y: #second.
31 g add: d.
32
33 d2 := RTData new.
34 d2 noDot.
35 d2 connectColor: Color red.
36 d2 points: (-15 to: 15 by: 0.1).
37 d2 y: f.
38 d2 x: #yourself.
39 g add: d2.
40 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:

```
1 r := Random new seed: 42.
2 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 (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.

2.9 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 := Perceptron 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
```

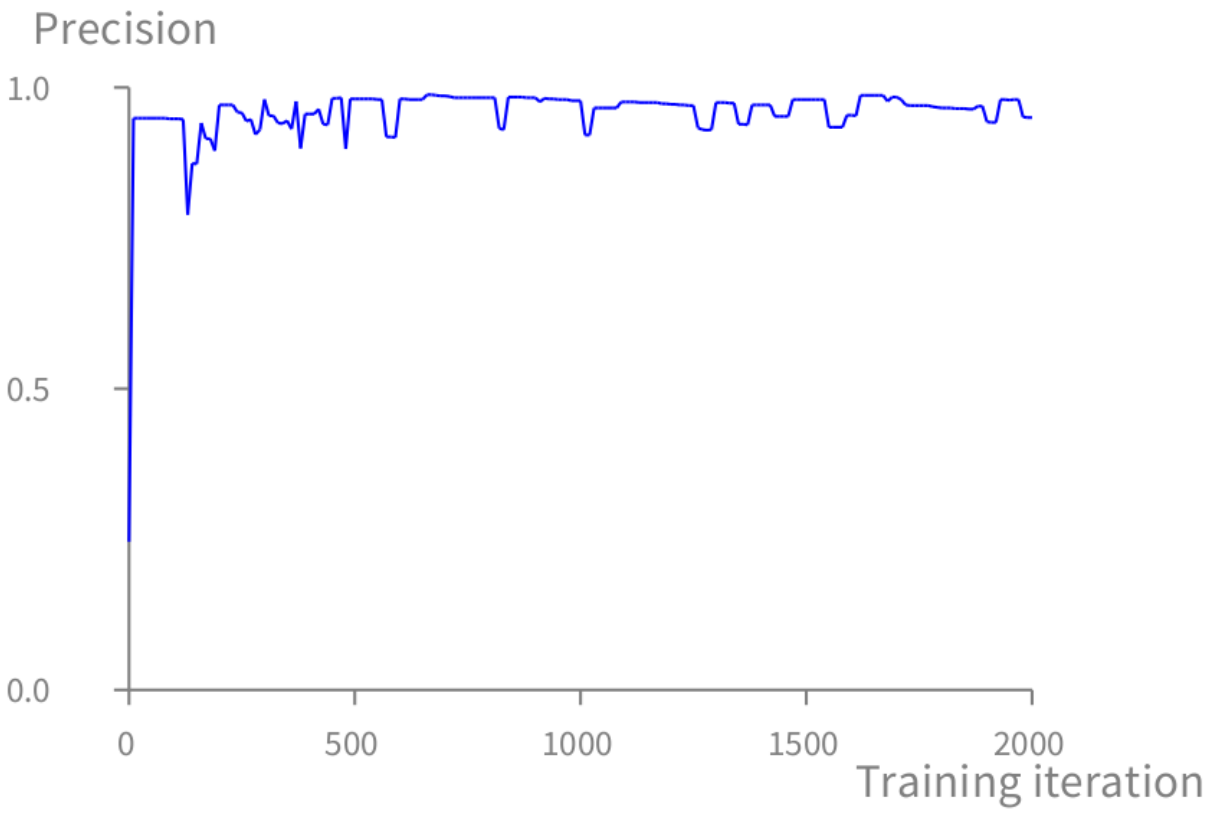

Figure 2.12: *perceptronPrecision*

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

2.10 Learning rate

What if we increase the learning rate?

2.11 What we have seen

- Providing the concept of perceptron
- A step-by-step guide on programming with Pharo
- Implemented a perceptron
- Teaching a perceptron

2.12 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?

2.13 Further reading

- Agile Programming

3 Artificial Neuron

In the previous chapter we have seen how a perceptron operates and how a simple learning algorithm can be implemented.

3.1 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. - A chain of perceptrons cannot learn.

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 perceptrons is that a small variation of z can produce a large variation of the output (going from 1 to 0, or from 0 to 1).

Learning algorithms that are commonly employed in neural networks require the following: 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.

3.2 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.

3.3 The Sigmoid Neuron

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