Agile Artificial Intelligence in Pharo

Implementing Neural Networks, Genetic Algorithms, and Neuroevolution

Alexandre Bergel

Apress®

Agile Artificial Intelligence in Pharo

Implementing Neural Networks, Genetic Algorithms, and Neuroevolution

Alexandre Bergel

Apress®

Agile Artificial Intelligence in Pharo: Implementing Neural Networks, Genetic Algorithms, and Neuroevolution

Alexandre Bergel
Santiago, Chile

ISBN-13 (pbk): 978-1-4842-5383-0 ISBN-13 (electronic): 978-1-4842-5384-7
https://doi.org/10.1007/978-1-4842-5384-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484253830. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Alexandre Bergel, Ph.D., is a professor (associate) at the Department of Computer Science (DCC), at the University of Chile and is a member of the Intelligent Software Construction laboratory (ISCLab). His research interests include software engineering, software performance, software visualization, programming environment, and machine learning. He is interested in improving the way we build and maintain software. His current hypotheses are validated using rigorous empirical methodologies. To make his research artifacts useful not only to stack papers, he co-founded Object Profile.

About the Technical Reviewer

Jason Whitehorn is an experienced entrepreneur and software developer and has helped many companies automate and enhance their business solutions through data synchronization, SaaS architecture, and machine learning. Jason obtained his Bachelor of Science in Computer Science from Arkansas State University, but he traces his passion for development back many years before then, having first taught himself to program BASIC on his family's computer while in middle school.

When he's not mentoring and helping his team at work, writing, or pursuing one of his many side-projects, Jason enjoys spending time with his wife and four children and living in the Tulsa, Oklahoma region. More information about Jason can be found on his website: https://jason.whitehorn.us.

Acknowledgments

Agile Artificial Intelligence in Pharo is the result of a long and collective effort made by the ESUG community and beyond. The writing and the necessary research of the book was sponsored by Lam Research and ESUG. Thank you! You made the book happen!

Many people helped get the book in shape. In no particular order, we are deeply grateful to CH Huang, Chris Thorgrimsson, Milton Mamani, Jhonny Cerezo, Oleks Zaytsev, Stéphane Ducasse, Torsten Bergmann, Serge Stinckwich, Alexandre Rousseau, Sean P. DeNigris, Julián Grigera, Cesar Rabak, Yvan Guemkam, John Borden, Sudhakar Krishnamachari, Leandro Caniglia, mldavis99, darth-cheney, Andy S., Jon Paynter, Esteban Lorenzano, Juan-Pablo Silva, Francisco Ary Martins, Norbert Fortelny, forty, Sebastián Zapata, and Renato Cerro.

Publishing a book involves some legal aspects that need to be carefully considered. We thank Fernanda Carvajal Gezan and Rosa Leal, from the University of Chile.

We also thank the Apress team for trusting in the project and thank Jason Whitehorn, who tech-reviewed the book.

Introduction

Artificial Intelligence (AI) is radically changing the way we use computers to solve problems. For example, by exploiting previous experience, which may be expressed in terms of examples, a machine can identify patterns in a given situation and try to identify the same patterns in a slightly different situation. This is essentially the way AI is used nowadays. The field of AI is moving quickly, and unfortunately, it is often difficult to understand.

The objective of the *Agile Artificial Intelligence in Pharo* book is to provide a practical foundation for a set of expressive artificial intelligence algorithms using the Pharo programming language. The book makes two large contributions over existing related books. The first contribution is to bring agility in the way some techniques related to artificial intelligence are designed, implemented, and evaluated. The book provides material in an incremental fashion, beginning with a little perception and ending with a full implementation of two algorithms for neuroevolution.

The second contribution is about making these techniques accessible to programmers by detailing their implementation without overwhelming the reader with mathematical material. There is often a significant gap between reading mathematical formulas and producing executable source code from those formulas, unfortunately. The book is meant to be accessible to a large audience by focusing on executable source code.

Overall, this book details and illustrates some easy-to-use recipes to solve actual problems. Furthermore, it highlights some technical details of these recipes using the Pharo programming language. *Agile Artificial Intelligence in Pharo* is not a book about how to use an existing API provided by external libraries. Instead, this book guides you to build your own API for artificial intelligence.

Book Overview

Agile Artificial Intelligence in Pharo is divided into three parts, each targeting a specific topic within the field of artificial intelligence—neural networks, genetic algorithms, and neuroevolution.

The first part of the book is about *neural networks*. A neural network is a computational metaphor simulating the interaction occurring between biological neurons. The chapter begins with the implementation of a single neuron and shows its limitations in terms of what it can achieve. Neural networks are then presented to solve more complex problems. Various examples involving relatively simple data classification tasks are presented.

The second part of the book covers *genetic algorithms* (GAs). The GA is a computational metaphor simulating the evolution occurring in biological species. GAs provide a way to solve problems without knowing the structure and shape of the solution in advance. GAs simulate the way biological species evolve over time. For two candidate solutions, as soon as the machine is able to say which one is closer to the solution, then GAs may be considered to solve the problem. Numerous examples are provided in this second part of the book, including an implementation of zoomorphic creatures, which is a simulation of artificial life. We define a zoomorphic creature as an artificial organism able to evolve in order to move itself through obstacles.

The third part of the book covers the field of *neuroevolution*, which is a combination of genetic algorithms and neural networks. The evolution of neural networks is called neuroevolution. Instead of *training a* neural network, as in classical deep learning (Part 1 of the book), neuroevolution begins with extremely simple networks and incrementally adds complexity to them. Evolution makes those networks able to solve particular tasks. This third part uses a Mario Bros-like game, which is used to build an artificial player using neuroevolution.

Installing Pharo

Pharo works on the three common platforms, Mac OSX, Windows, and Linux. The web page at `https://pharo.org/download` gives a very detailed instruction set and some links to download Pharo. Pharo is easy to install. Just a matter of a few clicks.

The content of the book is known to work up until Pharo 9. The code provided in the book does not heavily rely on the Pharo runtime. So the code provided in this book should be easy to adapt to future versions of Pharo or to another dialect of Smalltalk.

Accompanying Source Code

Agile Artificial Intelligence in Pharo is a book about programming. It provides and details a sizable amount of source code. Most of the code in the book is self-contained. This means that no external libraries are used besides the Pharo core and the Roassal visualization engine. Roassal is used to visually explore data and build a user interface. Readers may prefer to transcribe the code into Pharo or use our dedicated Git repository at https://github.com/Apress/agile-ai-in-pharo.

A script that begins with ellipses (i.e., `...`) means that you need to append the script to the last one seen before.

The code provided in this book is known to run on Pharo 8 and 9. To load the code, you simply need to open a playground and execute the following code:

```
Metacello new
        baseline: 'AgileArtificialIntelligence';
        repository: 'github://Apress/agile-ai-in-pharo/src';
        load.
```

The GitHub repository contains the `scripts` folder, which contains all the scripts and code snippets provided in the book.

The book focuses on Pharo; however, at the cost of a few small adaptations, all the provided code will run on an alternative Smalltalk implementation (e.g., VisualWorks).

Who Should Read This Book?

This book is designed to be read by a wide audience of programmers. As such, there is no need to have prior knowledge of neural networks, genetic algorithms, or neuroevolution. There is even no need to have a strong mathematical background. We made sure that there is no such prerequisite for most of the chapters. Some chapters require mild mathematical knowledge. However, these chapters are self-contained and skipping them will not negatively affect your overall understanding.

The book exposes some sophisticated AI techniques through the lenses of Pharo. Readers will acquire the theoretical and practical tools to be used in Pharo. Note that people willing to learn Pharo through AI are encouraged to complement it with additional sources of information.

The book is not made for people who want to learn about AI techniques without heavily investing in a programming activity. Instead, *Agile Artificial Intelligence in Pharo* is made for programmers who are either familiar with Pharo or are willing to be.

The Pharo Experience

The code provided in this book uses the Pharo programming environment. Programming in Pharo is a fantastic and emotional experience. Literally. Pharo gives meaning to Agile programming that cannot be experienced in another programming language, or at least, not to the same degree. We will try to convey this wonderful experience to the readers.

Pharo has a very simple syntax, which means that code should be understandable as soon as you have some programming knowledge. Chapter 2 briefly introduces the Pharo programming language and its environment in case you want to be familiar with it.

Why did we pick Pharo for this book? Pharo is a beautiful programming language with a sophisticated environment. It also provides a new way of communication between a human and a machine. By offering a live programming environment and a language with minimal syntax, programmers may express their thoughts in an incremental and open fashion. Although a number of similar programming environments exist (e.g., Scratch and Squeak), Pharo is designed to be used in an industrial software development setting.

Pharo syntax is concise, simple, unambiguous, and requires very little explanation to be fully understood. If you do not know Pharo, we encourage you to become familiar with the basics of its syntax and programming environment. Chapter 2 should help in that respect. The debugger, inspector, and playground are unrivaled compared to other programming languages and environments. Using these tools really brings an unmatchable feeling when programming.

Additional Reading

Agile Artificial Intelligence in Pharo provides a gentle introduction to Pharo in Chapter 2. However, the presentation of Pharo is shallow and not complete. Readers who do not know Pharo and but have experience in programming may find the chapter to be enough. Readers who want to deepen their knowledge may want to look for additional sources of learning. Here are some good readings on Pharo:

- `http://pharo.org` is the official website about Pharo.
- `https://mooc.pharo.org` is probably the most popular way to learn Pharo. It provides many short videos covering various aspects of Pharo.
- `http://books.pharo.org` offers many valuable books and booklets on Pharo.
- `http://agilevisualization.com` describes the Roassal visualization engine, which also contains a gentle introduction to Pharo.

Visualization is omnipresent in *Agile Artificial Intelligence in Pharo*. Roassal is used in many chapters and the reader is welcome to read *Agile Visualization* to become familiar with this wonderful visualization toolkit.

PART I

Neural Networks

CHAPTER 1

The Perceptron Model

All major animal groups have brains made of neurons. A *neuron* is a specialized cell that transmits electrochemical stimulation using an axon to other neurons. A neuron receives this nerve impulse via a *dendrite*. Since the early age of computers, scientists have tried to produce a computational model of a neuron. The perceptron was one of the first models to mimic the behavior of a neuron.

This chapter plays two essential roles in the book. First, it presents the *perceptron*, a fundamental model on which neural networks are based. Second, it also provides a gentle introduction to the Pharo programming language. The chapter builds a simple perceptron model in Pharo.

1.1 Perceptron as a Kind of Neuron

A perceptron is a kind of artificial neuron that models the behavior of a biological neuron. A perceptron is a machine that produces an output for a provided set of input values. Figure 1-1 gives a visual representation of a perceptron.

A perceptron accepts one, two, or more numerical values as inputs. It produces a numerical value as output (the result of a simple equation that we will see shortly). A perceptron operates on numbers, which means that the inputs and the output are numerical values (e.g., integers or floating point values).

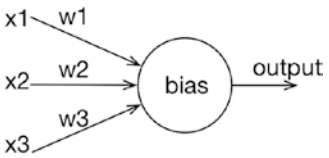

Figure 1-1. *Representing the perceptron*

A. Bergel, *Agile Artificial Intelligence in Pharo*, https://doi.org/10.1007/978-1-4842-5384-7_1

Figure 1-1 depicts a perceptron. A perceptron is usually represented as a circle with some inputs and one output. Inputs are represented as incoming arrows located on the left of the central circle and the output as an outgoing arrow on the right of it. The perceptron in Figure 1-1 has three inputs, noted `x1`, `x2`, and `x3`.

Not all inputs have the same importance for the perceptron. For example, an input may be more important than other inputs. Relevance of an input is expressed using a weight (also a numerical value) associated with that input. In Figure 1-1, the input `x1` is associated with the weight `w1`, `x2` with the weight `w2`, and `x3` with `w3`. Different relevancies of some inputs allow the network to model a specialized behavior. For example, for an image-recognition task, pixels located at the border of the picture usually have less relevance than the pixels located in the middle. Weights associated with the inputs corresponding to the border pixels will therefore be rather close to zero. In addition to the weighted input value, a perceptron requires a *bias*, a numerical value acting as a threshold. We denote the bias as `b`.

A perceptron receives a stimulus as input and responds to that stimulus by producing an output value. The output obeys a very simple rule: if the sum of the weighted inputs is above a particular given value, then the perceptron fires `1`; otherwise, it fires `0`. Programmatically, we first compute the sum of the weighted inputs and the bias. If this sum is strictly above `0`, then the perceptron produces `1`; otherwise, it produces `0`.

Formally, based on the perceptron given in Figure 1-1, we write `z = x1 * w1 + x2 * w2 + x3 * w3 + b`. In the general case, we write $z = \Sigma_i x_i * w_i + b$. The variable `i` ranges over all the inputs of the perceptron. If $z > 0$, then the perceptron produces `1` or if $z \leq 0$, it produces `0`.

In the next section, we will implement a perceptron model that is both extensible and maintainable. You may wonder what the big deal is. After all, the perceptron model may be implemented in a few lines of code. However, implementing the perceptron functionality is just a fraction of the job. Creating a perceptron model that is testable, well tested, and extensible is the real value of this chapter. Soon will see how to train a network of artificial neurons, and it is important to build this network framework on a solid base.

1.2 Implementing the Perceptron

In this section, we will put our hands to work and implement the perceptron model in the Pharo programming language. We will produce an object-oriented implementation of the model. We will implement a class called `Neuron` in a package called `NeuralNetwork`. The class will have a method called `feed`, which will be used to compute two values—`z` and the perceptron output.

This code will be contained in a package. To create a new package, you first need to open a *system browser* by selecting the corresponding entry in the Pharo menu. The system browser is an essential tool in Pharo. It allows us to read and write code. Most of the programming activity in Pharo typically happens in a system browser.

Figure 1-2 shows a system browser, which is composed of five different parts. The top part is composed of four lists. The left-most list gives the available and ready-to-be-used packages. In Figure 1-2, the names `Announcement`, `AST-Core` and `Alien` are examples of packages. The `Announcement` package is selected in the figure.

The second list gives the classes that belong to the selected package. Many classes are part of the `Announcement` package, including the classes called `Announcement`, `AnnouncementSet`, and `Announcer`.

The third list shows the method categories of the selected class. Method categories sort methods into logical groups to clarify their purpose and make them easier to find. Think of them as a kind of package for methods. Since no class is selected in the figure, no method category is listed.

The right-most list shows the methods of the selected class, filtered by the selected method category if any. Since no class is selected, no methods are listed. The bottom part of a system browser displays source code, which is one of the following:

Selection	Code Displayed
Method	Selected method source code
Class	Selected class definition
None	New class template

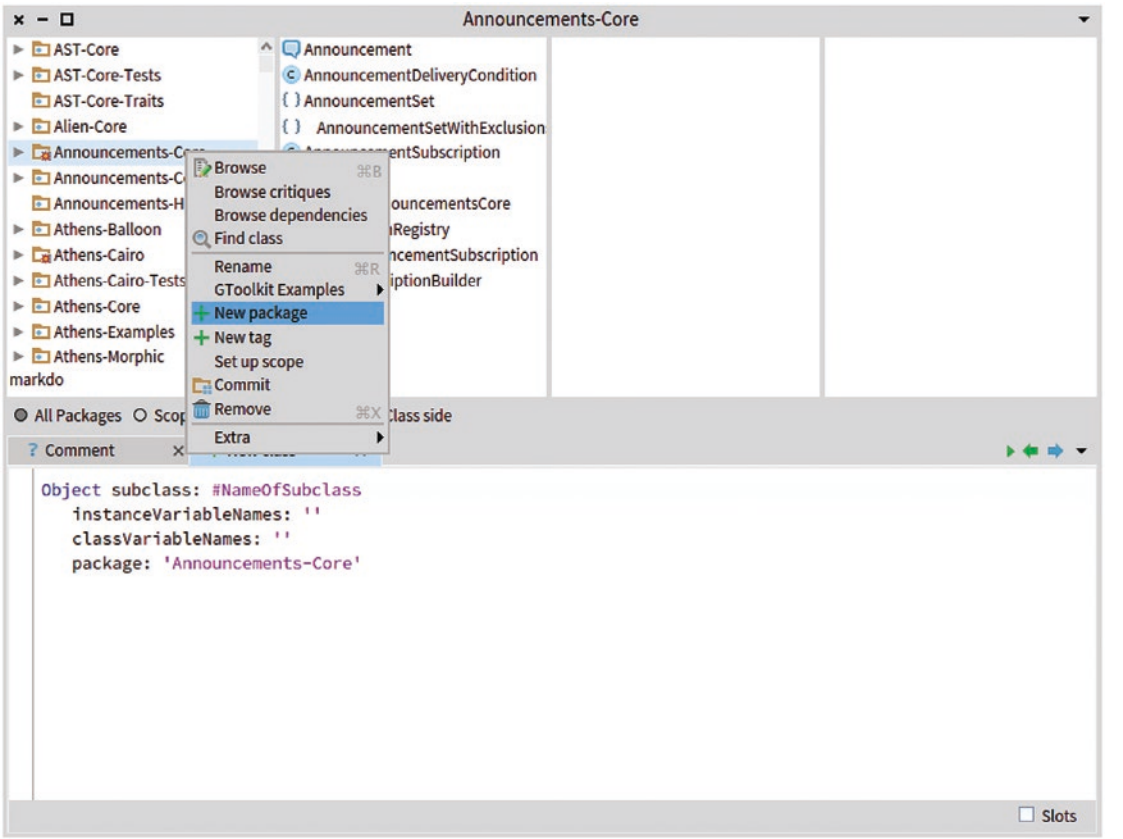

Figure 1-2. *The Pharo system browser*

Right-click the left-most top list to create a new package, named `NeuralNetwork`. This package will contain most of the code we will write in this first part of the book.

Select the package `NeuralNetwork` you just created and modify the template in the bottom pane as follows:

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

You then need to compile the code by "accepting" the source code. Right-click the text pane and select the `Accept` option. The `Neuron` class defines two instance variables—`weights` and `bias`. Note that we do not need to have variables for the inputs and output values. These values will be provided as message arguments and

returned values. We need to add some methods to define the logic of this perceptron. In particular, we need to compute the intermediate z and the output values. Let's first focus on the weights variable. We will define two methods to write a value in that variable and another one to read from it.

You may wonder why we define a class called Neuron and not Perceptron. In the next chapter, we will extend the Neuron class by turning it into an open abstraction for an artificial neuron. This Neuron class is therefore a placeholder for improvements we will make in the subsequent chapters. In this chapter we consider a perceptron, but in the coming chapter we will move toward an abstract neuron implementation. The name Neuron is therefore better suited.

Here is the code of the weights: method defined in the Neuron class:

```
Neuron>>weights: someWeightsAsNumbers
    "Set the weights of the neuron.
    Takes a collection of numbers as argument."
    weights := someWeightsAsNumbers
```

To define this method, you need to select the Neuron class in the class panel (second top list panel). Then, write the code given *without* Neuron>>, which is often prepended in documentation to indicate the class that should host the method. It is not needed in the browser because the class is selected in the top pane. Figure 1-3 illustrates this. Next, you should accept the code (again by right-clicking the Accept menu item). In Pharo jargon, accepting a method has the effect of actually compiling it (i.e., using the Pharo compiler to translate the Pharo source code into some bytecodes understandable by the Pharo virtual machine). Once it's compiled, a method may be executed. The code defines the method named called weights: which accepts one argument, provided as a variable named someWeightsAsNumbers.

The weights := someWeightsAsNumbers expression assigns the value someWeightsAsNumbers to the variable weights.

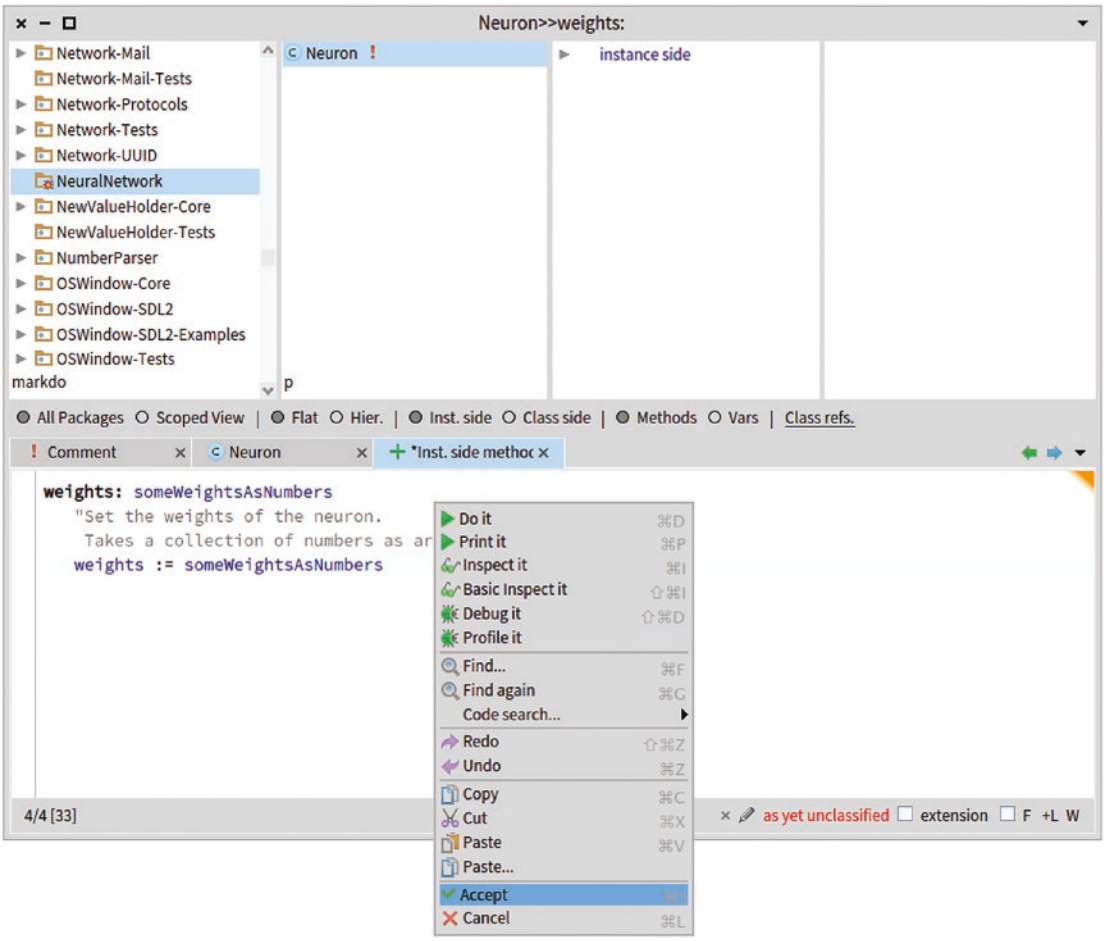

Figure 1-3. *The* `weights:` *method of the* `Neuron` *class*

Your system browser should now look like Figure 1-3. The `weights:` method writes a value to the variable `weights`. Its sibling method that returns the weight is

```
Neuron>>weights
    "Return the weights of the neuron."
    ^ weights
```

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

Similarly, you need to define methods to assign a value to the `bias` variable and to read its content. The method `bias:` is defined as follows:

```
Neuron>>bias: aNumber
    "Set the bias of the neuron"
    bias := aNumber
```

Reading the variable `bias` is provided by the following:

```
Neuron>>bias
    "Return the bias of the neuron"
    ^ bias
```

So far, we have defined the `Neuron` class, which contains two variables (`weights` and `bias`), and four methods (`weights:`, `weights`, `bias:`, and `bias`). We now need to define the logic of this perceptron by applying a set of input values and obtaining the output value. Let's add a `feed:` method that does exactly this small computation:

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

The `feed:` method simply translates the mathematical perceptron activation formula previously discussed into the Pharo programming language. The expression `inputs with: weights collect: [ :x :w | x * w ]` transforms the `inputs` and `weights` collections using the supplied function. Consider the following example:

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

The expression `#(1 2 3)` is an array made of three numbers—`1`, `2`, and `3`. The expression evaluates to `#(11 22 33)`. Syntactically, the expression means that the literal value `#(1 2 3)` receives a message called `with:collect:` with two arguments, the literal array `#(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 `Tools` top menu). A playground is a kind of command terminal for Pharo (e.g., `xterm` in the UNIX world). Figure 1-4 illustrates the evaluation of the expression (evaluated either by choosing Print It from the right-click menu or using the adequate shortcut—Cmd+p on OSX or Alt+p on other operating systems).

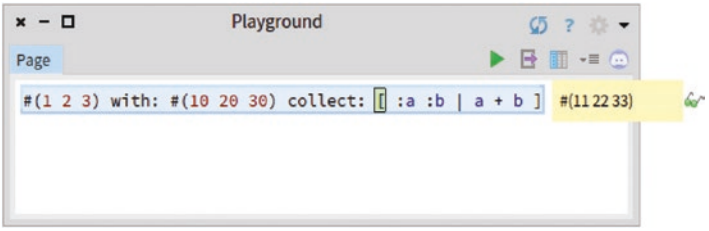

Figure 1-4. *The playground*

We can now play a little bit with the perceptron and evaluate the following code in the playground we just opened:

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

This piece of code evaluates to `1` (since `(5*1 + 2*2)-2` equals to `7`, which is greater than `0`), as shown in Figure 1-5.

Figure 1-5. *Evaluating the perceptron*

1.3 Testing the Code

Now it is time to talk about testing. Testing is an essential activity whenever we write code using Agile methodologies. Testing is about raising the confidence that the code we write does what it is supposed to do.

Although this book is not about writing large software artifacts, we *do* write source code. And making sure that this code can be tested in an automatic fashion significantly improves the quality of our work. More importantly, most code is read far more

often than it is written. Testing helps us produce maintainable and adaptable code. Throughout this book, we will improve our code base. It is very important to make sure that our improvements do not break existing functionalities.

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

We will now leave the playground for a while and return to the system browser to define a class called `PerceptronTest`:

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

The `TestCase` class belongs to the built-in Pharo code base. Subclassing it is the first step to creating a unit test. Many perceptrons will be created by the tests we define. We can define the method as follows:

```
PerceptronTest>>newNeuron
    "Return a new neuron"
    ^ Neuron new
```

Tests can now be added to `PerceptronTest`. Define the following method:

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

The `testSmallExample` method tests that the code snippet we previously gave returns the value `1`. You can run the test by clicking the gray circle located next to the method name. The gray circle turns green to indicate that the test passes (see Figure 1-6).

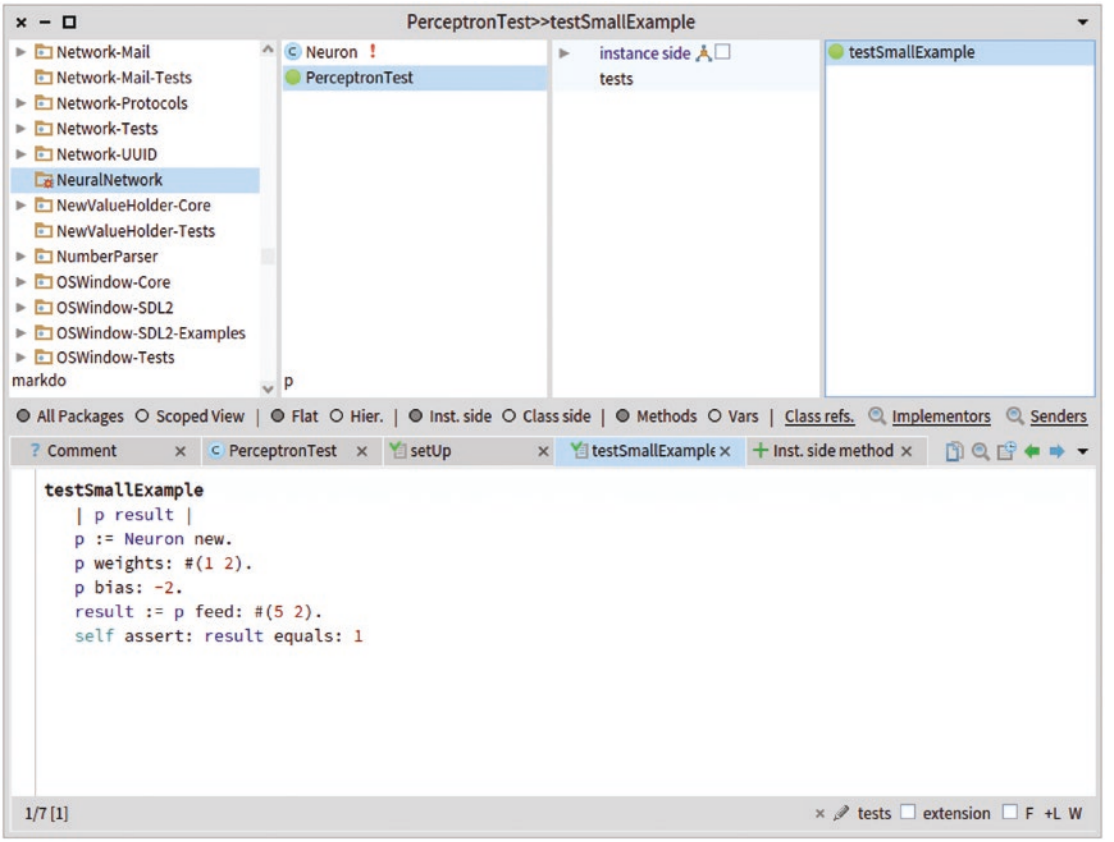

Figure 1-6. *Testing the perceptron*

A green test means that no assertion failed and no error was raised during the test execution. The `testSmallExample` method sends the `assert:equals:` message, which tests whether the first argument equals the second argument.

EXERCISE: So far, you have only shallowly tested this perceptron. You can improve these tests in two ways:

- Expand `testSmallExample` by feeding the perceptron `p` with different values (e.g., `-2` and `2` gives `0` as a result).
- Test the perceptron with different weights and biases.

In general, it is a very good practice to write a thorough suite of tests, even for a small component such as this `Neuron` class.

1.4 Formulating Logical Expressions

A canonical example of using a perceptron 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 the modeled 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. Recall that `#(1 1)` is an array of size `2` that contains the number `1` twice. The AND gate is a basic digital logic gate, and it is an idealized device for implementing the AND boolean function. The AND gate may be represented as the following table:

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

We could therefore verify this with a new test method:

```
PerceptronTest>>testAND
    | p |
    p := self newNeuron.
    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.
```