# Agile Artificial Intelligence

Alexandre Bergel
RelationalAI, Switzerland

https://bergel.eu

Agile Artificial Intelligence in Pharo
Implementing Neural Networks, Genetic Algorithms, and Neuroevolution
Alexandre Bergel
Apress®

Part 1: Neural network

+

Part 2: Genetic algorithm

=

Part 3: Neuroevolution

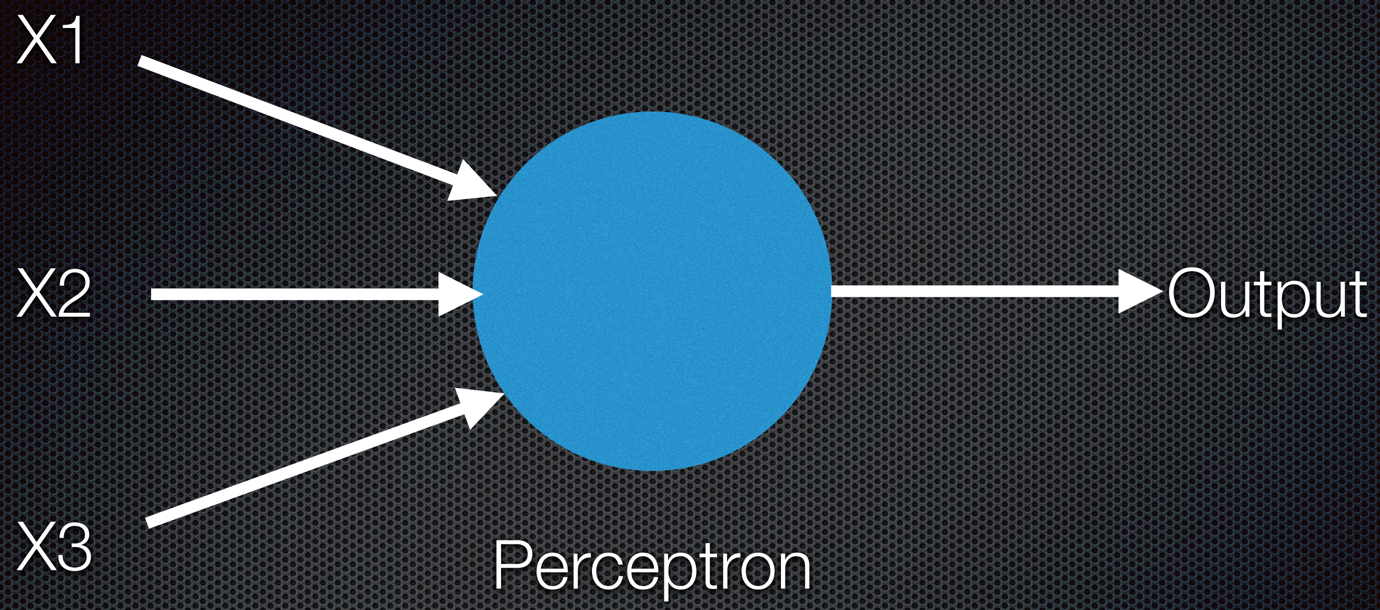
Part 1: Neural network

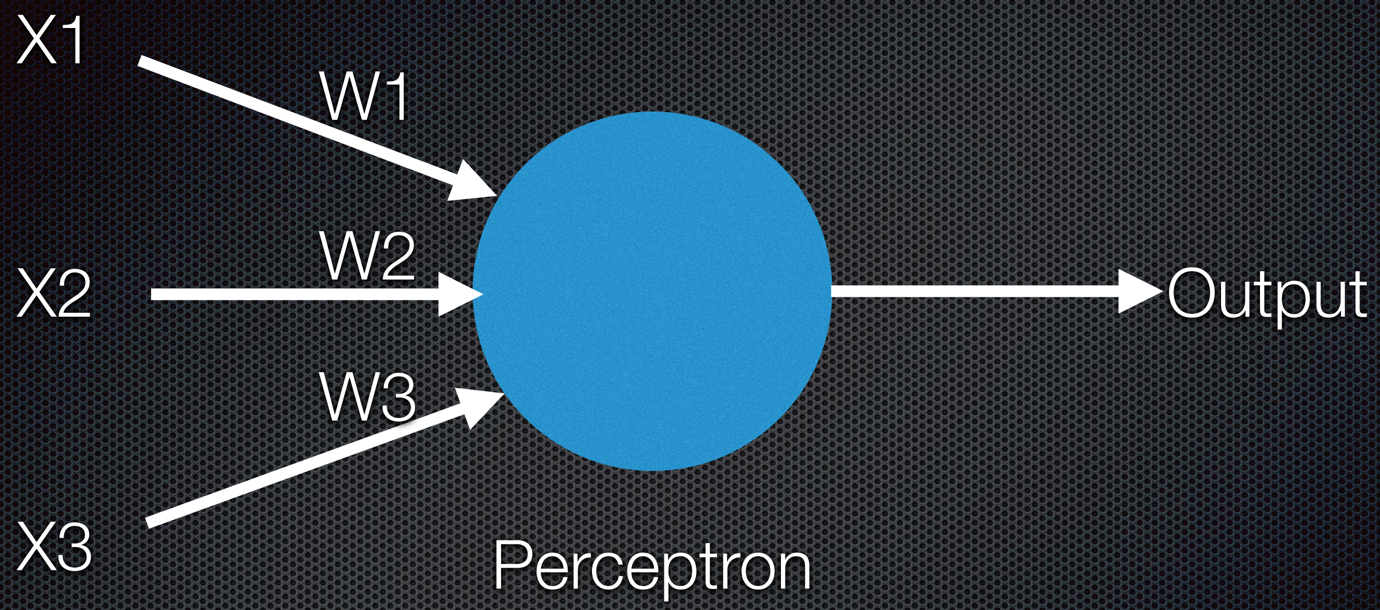
+

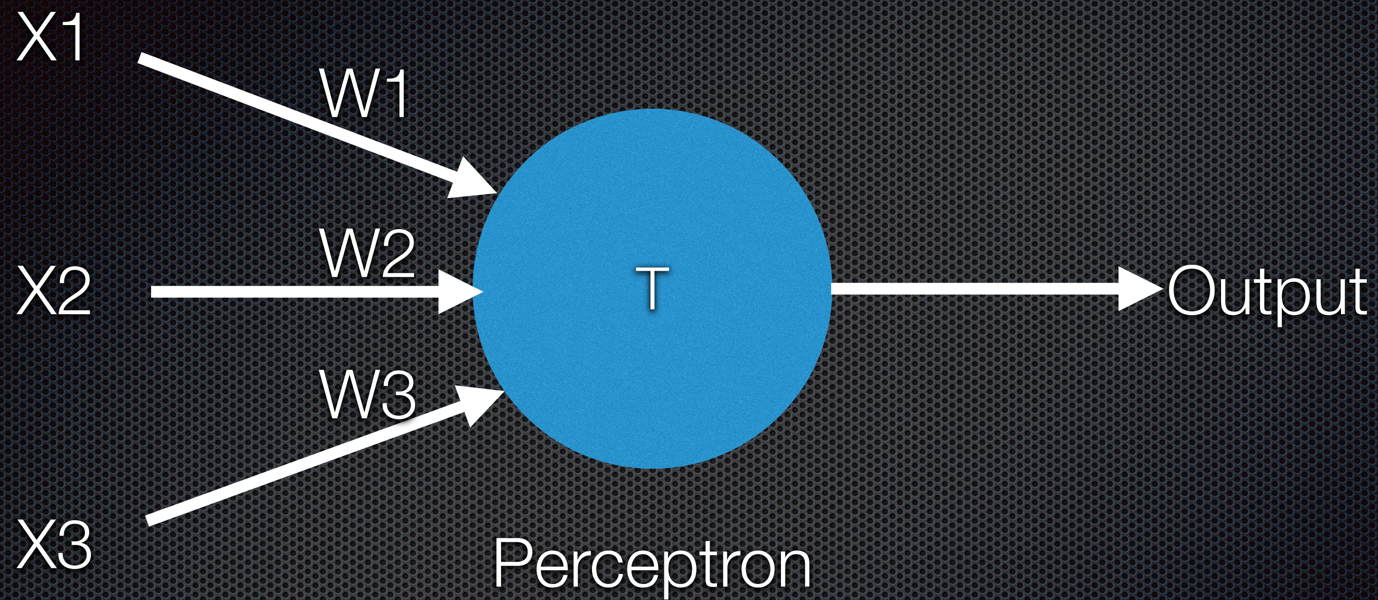
Part 2: Genetic algorithm

=

Part 3: Neuroevolution


X1
X2
X3
Output
Perceptron


X1
W1
X2
W2
W3
X3
Output
Perceptron



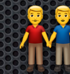

```
Output := (X1*W1) + (X2*W2) + (X3*W3) > T ifTrue: [ 1 ] ifFalse: [ 0 ]
```

# Perceptron 🤘

- Great **heavy metal** concert this week end
- As a metal-lover, you would like to go
- Three factors:
  - X1: Is the weather good?
  - X2: Does someone accompany me?
  - X3: Can I go without a car?

# Perceptron 🤘

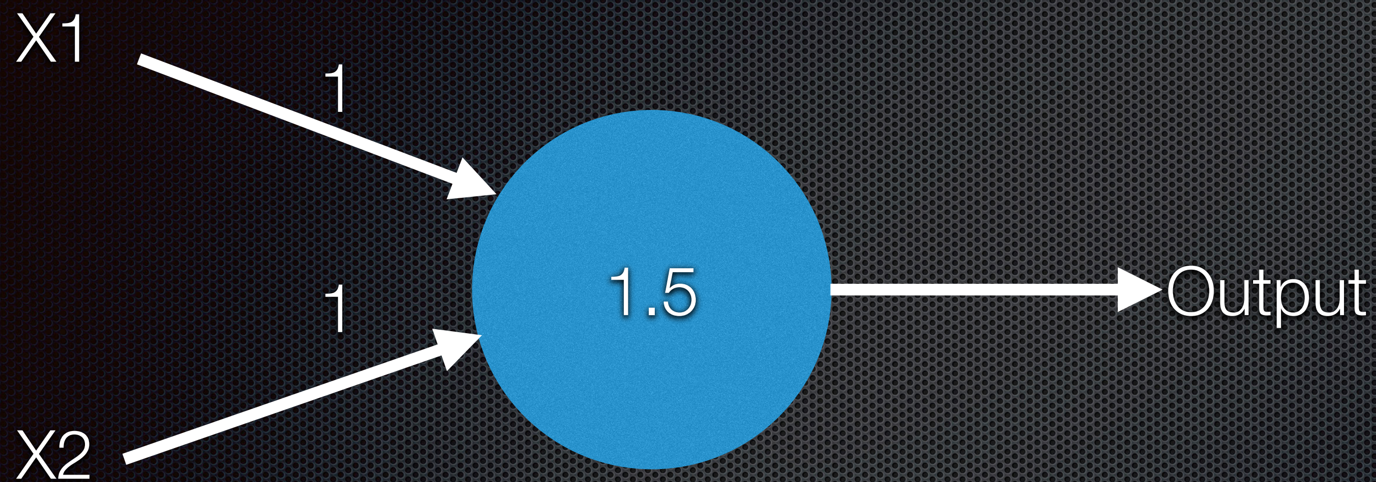
- Great heavy metal concert this week end
- As a metal-lover, you would like to go
- Three factors:
  - X1: Is the weather good?
  - X2: Does someone accompany me?
  - X3: Can I go without a car?



W1 = 2
W2 = 6
W3 = 2

W1 = 1
W2 = 1
W3 = 8

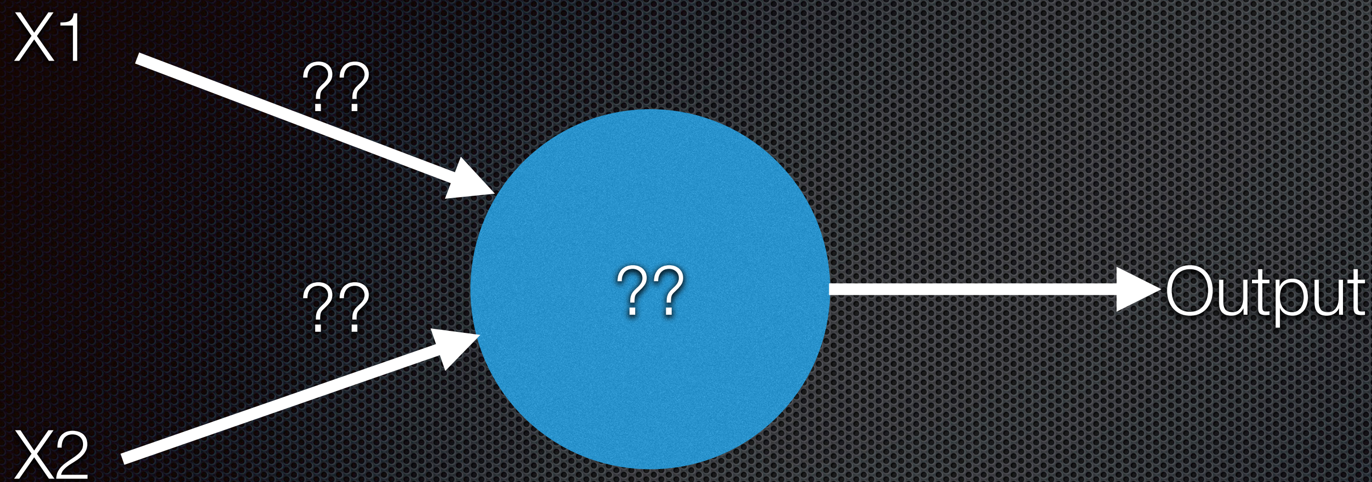
# AND Logical gate




| X1 | X2 | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

```
Output := (X1*W1) + (X2*W2) > T ifTrue: [ 1 ] ifFalse: [ 0 ]
```

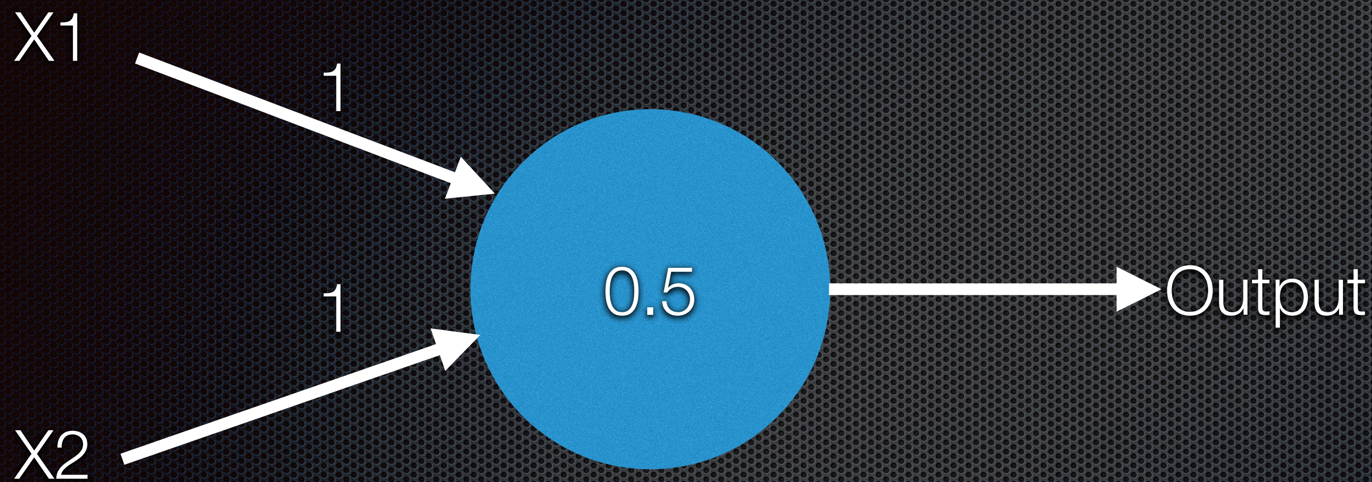
# OR Logical gate




| X1 | X2 | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

```
Output := (X1*W1) + (X2*W2) > T ifTrue: [ 1 ] ifFalse: [ 0 ]
```

# OR Logical gate




| X1 | X2 | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

```
Output := (X1*W1) + (X2*W2) > T ifTrue: [ 1 ] ifFalse: [ 0 ]
```

# Perceptron learning algorithm

```
diff = desiredOutput - realOutput
lr = 0.1
For all N:
   weightN = weightN + (lr * inputN * diff)
bias = bias + (lr * diff)
lr is called the learning rate
```

```
learningCurveNeuron := OrderedCollection new.
0 to: 1000 do: [ :nbOfTrained |
    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.
     learningCurveNeuron add: res / 4.
    ].
learningCurvePerceptron := OrderedCollection new.
0 to: 1000 do: [ :nbOfTrained |
   p := Neuron new.
   p step.
    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.
     learningCurvePerceptron add: res / 4.
    ].
g := RTGrapher new.
d := RTData new.
d label: 'Sigmoid neuron'.
```

CHAPTER 3

# Neural Networks

The previous chapter covered the design and implementation of an individual neuron. This chapter builds upon the effort initiated in previous chapters by connecting multiple neurons. We provide a complete implementation of a neural network and a backpropagation algorithm, which brings us to the core of the first part of the book.

## 3.1 General Architecture

An artificial neural network is a computing system inspired by the biological neural networks found in 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. In neural networks, just as with the perceptron and sigmoid neuron, knowledge is acquired through learning.

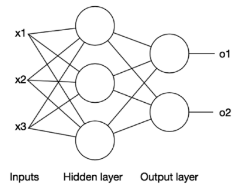



***Figure 3-1.*** *Example of a neural network*



© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence in Pharo*, https://doi.org/10.1007/978-1-4842-5384-7_3

CHAPTER 4

# Theory on Learning

Understanding the learning algorithm that's used with neural networks involves a fair dose of mathematical notations. This chapter details some relevant theoretical aspects of the way that neural networks operate. We will therefore review the notions of loss functions and gradient descent. Note that this chapter is by no means a complete description of how networks learn. As indicated at the end of this chapter, many other people have done an excellent job of accurately describing the theoretical foundation of learning and optimization mechanisms. Instead, this chapter is meant to back up some aspects of the implementation explained in the previous chapters, with the assumption that you are comfortable with basic differential calculus.

You can safely skip this chapter if the theory behind neural networks does not interest you.

This chapter intensively uses Roassal to visualize data. You therefore need to have it loaded, as indicated in the previous chapters, in order to run the scripts in this chapter.

## 4.1 Loss Function

A network needs to learn in order to reduce the amount of errors it makes when making a prediction. Such a prediction could be used either to classify data or to run regression analysis. It is therefore essential to have a way to measure the errors made by a network. This is exactly what a loss function does.

A *loss function* is a measure of the error made by a particular model. The loss function is also commonly called the *error function* or the *cost function*. To illustrate the use and need of a loss function, let's consider the following problem: for a given set of points, what is the straight line that is the closest to these points?

Consider a set of four points:

```
points :={(1 @ 3.0). (3 @ 5.2). (2 @ 4.1). (4 @ 7.5)}.
```



© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence in Pharo*, https://doi.org/10.1007/978-1-4842-5384-7_4

CHAPTER 5

# Data Classification

Neural networks have an incredibly large range of applications. Classifying data is a prominent one, and this chapter is devoted to it.

## 5.1 Training a Network

In the previous chapter, we saw that we can obtain a trained neural network to express the XOR logical gate. In particular, we saw the following script:

```
n := NNetwork new.
n configure: 2 hidden: 3 nbOfOutputs: 1.

20000 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).
].
```

After evaluating this script, the expression `n feed: #(1 0)` evaluates to `#(0.9530556769505442)`, which is an array having an expected float value close to `1`. If we step back a bit, we see that the script is actually very verbose. For example, why should we manually handle the repetition? Why is the message `train:desiredOutputs:` sent so many times? We can greatly simplify the way networks are trained by providing a bit of infrastructure.

Consider the following method:

```
NNetwork>>train: train nbEpochs: nbEpochs
        "Train the network using the train dataset."
        | sumError outputs expectedOutput epochPrecision t |
```



© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence in Pharo*, https://doi.org/10.1007/978-1-4842-5384-7_5

CHAPTER 3

# Neural Networks

The previous chapter covered the design and implementation of an individual neuron. This chapter builds upon the effort initiated in previous chapters by connecting multiple neurons. We provide a complete implementation of a neural network and a backpropagation algorithm, which brings us to the core of the first part of the book.

## 3.1 General Architecture

An artificial neural network is a computing system inspired by the biological neural networks found in 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 recognitio

neuron, knowledge is a

*Figure 3-1.* Example

© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intel*

CHAPTER 4

# Theory on Learning

Understanding the learning algorithm that's used with neural networks involves a fair dose of mathematical notations. This chapter details some relevant theoretical aspects of the way that neural networks operate. We will therefore review the notions of loss functions and gradient descent. Note that this chapter is by no means a complete description of how networks learn. As indicated at the end of this chapter, many other people have done an excellent job of accurately describing the theoretical foundation of learning and optimization mechanisms. Instead, this chapter is meant to back up some aspects of the implementation explained in the previous chapters, with the assumption that you are comfortable with basic differential calculus.

You can safely skip this chapter if the theory behind neural networks does not interest you.

This chapter intensively uses Roassal to visualize data. You therefore need to have it loaded, as indicated in the previous chapters, in order to run the scripts in this chapter.

## Loss Function

work needs to learn in order to reduce the a
diction. Such a prediction could be used eit
sis. It is therefore essential to have a way to
is exactly what a loss function does.
*loss function* is a measure of the error made
ion is also commonly called the *error functi*
nd need of a loss function, let's consider the
s, what is the straight line that is the closest
onsider a set of four points:

```
ts :={(1 @ 3.0). (3 @ 5.2). (2 @ 4.1)
```

andre Bergel 2020
gel, *Agile Artificial Intelligence in Pharo*, https://doi.org/1

CHAPTER 5

# Data Classification

Neural networks have an incredibly large range of applications. Classifying data is a prominent one, and this chapter is devoted to it.

## 5.1 Training a Network

In the previous chapter, we saw that we can obtain a trained neural network to express the XOR logical gate. In particular, we saw the following script:

```
n := NNetwork new.
n configure: 2 hidden: 3 nbOfOutputs: 1.

20000 timesRepeat: [
        n train: #(0 0) desiredOutputs: #(0).
        n train: #(0 1) desiredOutputs: #(1).
                        iredOutputs: #(1).
                        iredOutputs: #(0).
```

the expression n feed: #(1 0) evaluates to
ı is an array having an expected float value close to 1.
: the script is actually very verbose. For example, why
repetition? Why is the message train:desiredOutputs:
atly simplify the way networks are trained by providing a

hod:

```
ochs: nbEpochs
using the train dataset."
expectedOutput epochPrecision t |
```



haro, https://doi.org/10.1007/978-1-4842-5384-7_5

CHAPTER 6

# A Matrix Library

In the previous chapters, we presented an implementation of a neural network made of layers and neurons (i.e., instances of NeuronLayer and Neuron). Although instructive, that implementation does not reflect classical ways of implementing a neural network. A layer can be expressed as a matrix of weights and a vector of biases. This is how most libraries that build neural networks (e.g., TensorFlow and PyTorch) actually operate.

This chapter lays out a small library to build and manipulate matrices. This chapter is an important foundation for the subsequent chapter, which is about how networks can be implemented using matrices. Matrices are a particular data structure for which operations cannot efficiently be implemented in Pharo. We will write these costly operations in C but make them accessible within Pharo.

In addition to defining a matrix library, this chapter highlights one particular aspect of Pharo, which is the use of Foreign Function Interface (FFI). This is a relevant mechanism whenever one wishes to make Pharo use external libraries written using the C or C++ programming languages. For example, TensorFlow is written in C++, which may be accessed from Pharo using the very same technique presented in this chapter.

This chapter is long and contains many inter-dependent methods. The chapter needs to be fully implemented before being functional.

## 6.1 Matrix Operations in C

Pharo does not provide built-in features to manipulate matrices. Although we could implement them in Pharo, it would suffer from very poor performance. Instead, we will code a small library in C to support the elementary C operations. Create a file named matrix.c with the following C code:

```
void dot(double *m1, int m1_nb_rows, int m1_nb_columns, double *m2,
                    int m2_nb_rows, int m2_nb_columns,
                    double *res) {
```



© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence in Pharo*, https://doi.org/10.1007/978-1-4842-5384-7_6

CHAPTER 7

# Matrix-Based Neural Networks

This chapter revises the implementation of our neural network. In this revision, our network will use matrices to compute the forward and backward propagation algorithms. Overall, our matrix-based implementation is composed of two classes, NMLayer and NMNetwork. Since most of the computation is delegated to the matrix library we defined in the previous chapter, our new version of the neural network is rather light in terms of code.

## 7.1 Defining a Matrix-Based Layer

A neural network is composed of layers. We describe a layer as an instance of the NMLayer class, defined as follows:

```
Object subclass: #NMLayer
    instanceVariableNames: 'w b delta output previous next lr
       numberOfExamples'
    classVariableNames: ''
    package: 'NeuralNetwork-Matrix'
```

The NMLayer class does not contain neurons, as we saw in our first implementation. Instead, a matrix describing weights is used and kept in the w variable, and another matrix is used to keep the bias vector, kept in the b variable.

The initialization of a layer simply consists of setting the default learning rate:

```
NMLayer>>initialize
    super initialize.
    lr := 0.1
```



© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence in Pharo*, https://doi.org/10.1007/978-1-4842-5384-7_7

Part 1: Neural network

+

Part 2: Genetic algorithm

=

Part 3: Neuroevolution

Part 1: Neural network

+

Part 2: Genetic algorithm

=

Part 3: Neuroevolution

*“One general law, leading to the advancement of all organic beings, namely, vary, let the strongest live and the weakest die”*

*— Charles Darwin*

*“Guess the 3-letter word I have in mind”*

Secret word: cat

*"Guess the 3-letter word I have in mind"*

Secret word: cat

Secret word: cat

*gaz*

*cow*

*poz*

Best score = 1

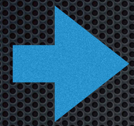
Secret word: cat

*gaz*

*cow*

*poz*

Applying genetic operators



*gow*

*caz*

*pow*

Best score = 1

Best score = 2

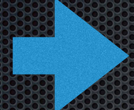
Secret word: cat

*gaz*

*cow*

*poz*

Applying genetic operators

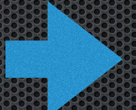


*gow*

*caz*

*pow*

Applying genetic operators



*goz*

*cat*

*poz*

Best score = 1

Best score = 2

Best score = 3

```
stringToFind := 'cat'.
g := GAEngine new.
g populationSize: 1000.
g numberOfGenes: stringToFind size.
g createGeneBlock: [ :rand :index :ind | ($a to: $z)
atRandom: rand ].
g fitnessBlock: [ :genes |
  (stringToFind asArray with: genes collect: [ :a :b |
    a = b ifTrue: [ 1 ] ifFalse: [ 0 ] ]) sum ].
g run.
g visualize open
```

C

A

T

Muscle length

Muscle force

Muscle tension

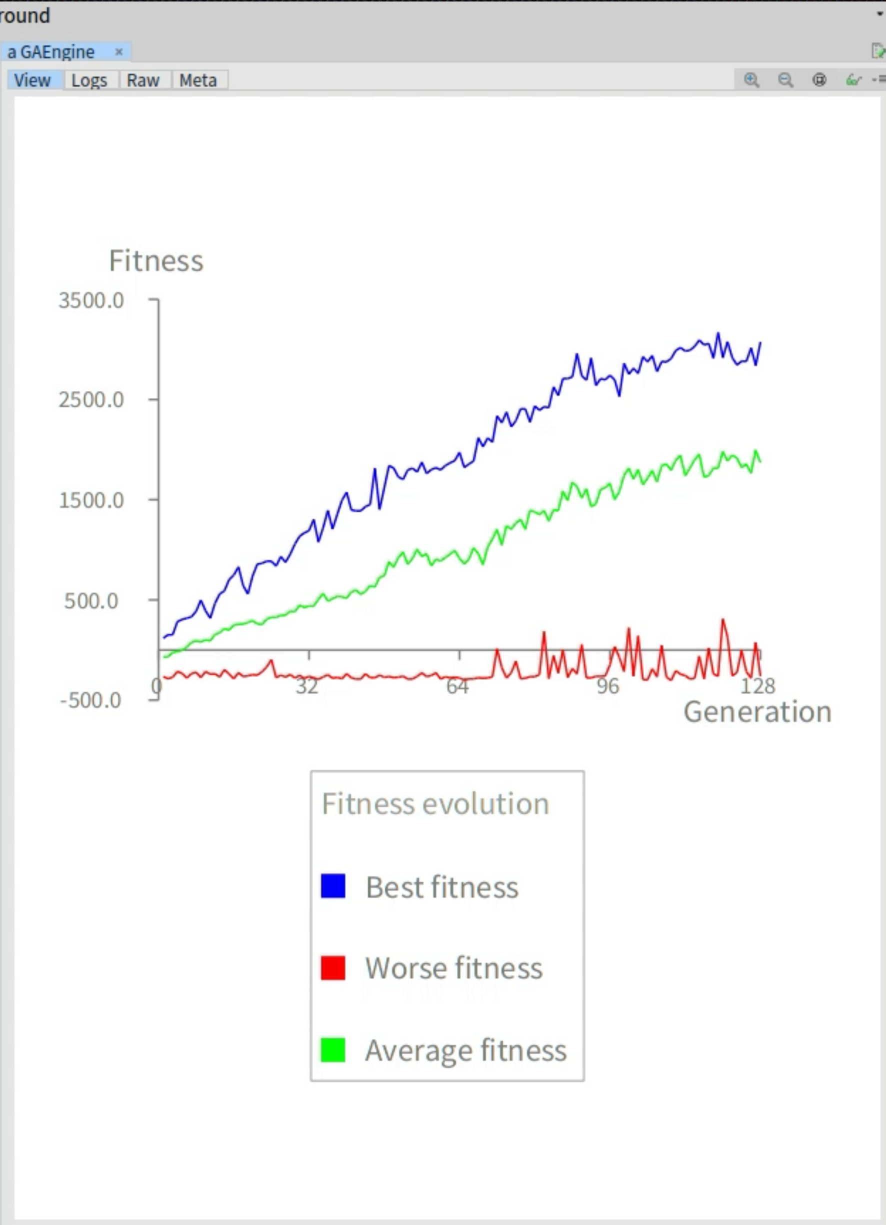
…

Playground

Page

```
g selection: (GATournamentSelection new).
g mutationRate: 0.02.
g endForMaxNumberOfGeneration: 128.
g populationSize: 100.
g numberOfGenes: numberOfMuscles * 5.
g createGeneBlock: [ :r :index | mg valueForIndex: index ].
g fitnessBlock: [ :genes |
   creature := CCreature new configureBall: numberOfNodes.
   creature materialize: genes.
   creature resetPosition.
   c := CWorld new.
   c addCreature: creature.
   1 to: 25 by: 3 do: [ :x |
      c addPlatform: (CPlatform new height: 20; width: 80; translateTo: x *
100 @ -10).
      c addPlatform: (CPlatform new height: 20; width: 80; translateTo: x *
100 + 50 @ -30).
      c addPlatform: (CPlatform new height: 20; width: 80; translateTo: x *
100 + 100 @ -50).
      c addPlatform: (CPlatform new height: 20; width: 80; translateTo: x *
100 + 150 @ -70).
   ].
   c addCreature: creature.
   3000 timesRepeat: [ c beat ].
   creature position x
].
g run.

creature := CCreature new configureBall: 10.
creature materialize: g result.
c := CWorld new.
1 to: 25 by: 3 do: [ :x |
   c addPlatform: (CPlatform new height: 20; width: 80; translateTo: x * 100
@ -10).
   c addPlatform: (CPlatform new height: 20; width: 80; translateTo: x * 100
+ 50 @ -30).
   c addPlatform: (CPlatform new height: 20; width: 80; translateTo: x * 100
+ 100 @ -50).
   c addPlatform: (CPlatform new height: 20; width: 80; translateTo: x * 100
+ 150 @ -70).
].
c addCreature: creature.
c open
```



Pharo Tools Roassal | System Debugging Windows Help

Playground

Page

```
   creature resetPosition.
   c := CWorld new.
   c addPlatform: (CPlatform new height: 20; width: 80; translateTo: 100 @
-10).
   c addPlatform: (CPlatform new height: 20; width: 80; translateTo: 400 @
-10).
   c addPlatform: (CPlatform new height: 20; width: 80; translateTo: 700 @
-10).
   c addPlatform: (CPlatform new height: 20; width: 80; translateTo: 1000 @
-10).
   c addCreature: creature.
   3000 timesRepeat: [ c beat ].
   creature position x
] value: (g logs at: 128) fittestIndividual genes.

c := CWorld new.
creature := CCreature new color: Color red; configureBall: 10.
creature materialize: g logs last fittestIndividual genes.
c addCreature: creature.

creature := CCreature new color: Color yellow darker darker; configureBall: 10.
creature materialize: (g logs at: 50) fittestIndividual genes.
c addCreature: creature.

creature := CCreature new color: Color blue darker darker; configureBall: 10.
creature materialize: (g logs at: 100) fittestIndividual genes.
c addCreature: creature.

creature := CCreature new color: Color green darker darker; configureBall: 10.
creature materialize: (g logs at: 90) fittestIndividual genes.
c addCreature: creature.

c addPlatform: (CPlatform new height: 20; width: 80; translateTo: 100 @ -10).
c addPlatform: (CPlatform new height: 20; width: 80; translateTo: 400 @ -10).
c addPlatform: (CPlatform new height: 20; width: 80; translateTo: 700 @ -10).
c addPlatform: (CPlatform new height: 20; width: 80; translateTo: 1000 @ -10).
c open
```

a GAEngine

View Logs Raw Meta

Fitness
4500.0
3500.0
2500.0
1500.0
500.0
-500.0
0 32 64 96 128
Generation

Fitness evolution
Best fitness
Worse fitness
Average fitness

CHAPTER 9

# Genetic Algorithms in Action

This chapter illustrates the use of genetic algorithms by solving a number of difficult algorithmic problems. Most of the problems presented in this chapter involve some arithmetic operations and therefore have a mathematical flavor.

## 9.1 Fundamental Theorem of Arithmetic

A prime number is a whole number greater than 1 whose only factors are 1 and itself. For example, 7 is a prime because it can only be divided by 7 and 1. The number 10 is not a prime because it can be divided by 2 and 5—two prime numbers.

In number theory, there is a theorem called *the fundamental theorem of arithmetic*, which states "any integer greater than 1 is either a prime number itself, or can be written as a unique product of prime numbers." Note that this representation is unique, except for the order of the factors. For example, the number 345 is a multiplication of factors 3*5*23. Finding this list of factors is computationally expensive. We will use genetic algorithms to identify the prime factors of any given number. As such, a gene will represent a prime number factor.

It is relevant to note that the number of factors depends on the number to be factored out. For example, the number `345` has three factors (`3`, `5`, and `23`), whereas the number `788,389` has four factors since `788,389 = 7 * 41 * 41 * 67`. In the genetic algorithm we presented in the previous chapter, all the individuals have the exact same number of genes. How do we represent an arbitrary number of genes then? One way that fits well with our situation is to consider `1` as a possible factor. Assuming each individual has 10 genes, the factors of `345` can be encoded with the values `3`, `5`, `23`, and seven times the factor `1`. The solution will then be the factors contained in an individual for which we ignore the value `1`.



© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence in Pharo*, https://doi.org/10.1007/978-1-4842-5384-7_9

CHAPTER 10

# The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a classical algorithm problem. It consists of identifying the shortest possible route between several connected cities. Not only is the problem relevant from an algorithmic point of view, but it also has many concrete applications, like microchip manufacturing, as you will shorty see.

The chapter incrementally builds a non-trivial solution to the problem using a genetic algorithm. The chapter begins with a naive approach to a robust, practical way of solving it.

## 10.1 Illustration of the Problem

***Figure 10-1.*** *Setup of the Traveling Salesman Problem*

Consider the example given in Figure 10-1. The figure shows four cities located in a horizontal diamond. Each city has a 2D coordinate and is therefore located in a two-dimensional plane. Assuming the traveler begins their journey at City A, many paths are possible to visit all the cities.



© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence in Pharo*, https://doi.org/10.1007/978-1-4842-5384-7_10

CHAPTER 11

# Exiting a Maze

Genetic algorithms are
This chapter applies a g
a simple situation (a rob
the distance between th
in a randomly generate

## 11.1 Encoding

We will model the maze
are fixed positions. The
at the bottom-right corr

Our robot will follov
north, south, west, or ea

Applied to our gene
at the maze entrance. Th
encoded in the genes, a
exit. The genetic algorit
is getting closer to the e

## 11.2 Robot De

The very first step is to r
knows its position and t

```
Object subclass: #GA
    instanceVariable
    classVariableNam
    package: 'Robot'
```

© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelli



```
map := GARobotMap new fillWithWalls: 80.
robot := GARobot new.
robot map: map.
g := GAEngine new.
g endIfNoImprovementFor: 5.
g numberOfGenes: 100.
g populationSize: 250.
g createGeneBlock: [ :rand :index :ind | #($N $S $W $E) atRandom: rand ].
g minimizeComparator.
g
    fitnessBlock: [ :genes |
        | path penalty |
        path := robot followOrders: genes.
        penalty := path size / 2.
        (robot position dist: map exitPosition) + penalty ].
g run.
map drawRobotPath: (robot followOrders: g result).
map open
```

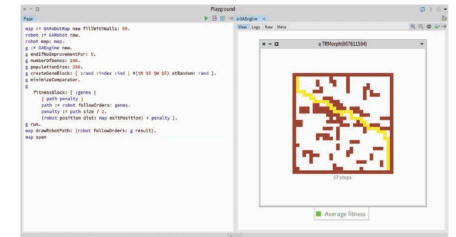


***Figure 11-3.*** *Short robot footprint*

CHAPTER 9

# Genetic Algorithms in Action

This chapter illustrates the use of genetic algorithms by solving a number of difficult algorithmic problems. Most of the problems presented in this chapter involve some arithmetic operations and therefore have a mathematical flavor.

## 9.1 Fundamental Theorem of Arithmetic

A prime number is a whole number greater than 1 whose only factors are 1 and itself. For example, 7 is a prime becau
prime because it can be div
In number theory, there
which states "any integer gr
as a unique product of prim
for the order of the factors. I
3*5*23. Finding this list of fa
algorithms to identify the pr
represent a prime number f
It is relevant to note tha
factored out. For example, t
the number 788,389 has fou
algorithm we presented in t
number of genes. How do w
fits well with our situation is
has 10 genes, the factors of i
the factor 1. The solution wi
ignore the value 1.

© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence*

CHAPTER 10

# The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a classical algorithm problem. It consists of identifying the shortest possible route between several connected cities. Not only is the problem relevant from an algorithmic point of view, but it also has many concrete applications, like microchip manufacturing, as you will shorty see.

The chapter incrementally builds a non-trivial solution to the problem using a genetic algorithm. The chapter begins with a naive approach to a robust, practical way of solving it.

## 10.1 Illustration of the Problem

***Figure 10-1.*** *Setup of the Trav*

Consider the example given in Fig
horizontal diamond. Each city ha
dimensional plane. Assuming the
possible to visit all the cities.

© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence in Pha*

CHAPTER 11

# Exiting a Maze

Genetic algorithms are
This chapter applies a g
a simple situation (a rob
the distance between th
in a randomly generate

## 11.1 Encoding

We will model the maze
are fixed positions. The
at the bottom-right corr
Our robot will follov
north, south, west, or ea
Applied to our gene
at the maze entrance. Th
encoded in the genes, a
exit. The genetic algorit
is getting closer to the e

## 11.2 Robot De

The very first step is to m
knows its position and t

```
Object subclass: #GA
    instanceVariable
    classVariableNam
    package: 'Robot'
```

© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelli*



```
map := GARobotMap new fillWithWalls: 80.
robot := GARobot new.
robot map: map.
g := GAEngine new.
g endIfNoImprovementFor: 5.
g numberOfGenes: 100.
g populationSize: 250.
g createGeneBlock: [ :rand :index :ind | #($N $S $W $E) atRandom: rand ].
g minimizeComparator.
g
    fitnessBlock: [ :genes |
        | path penalty |
        path := robot followOrders: genes.
        penalty := path size / 2.
        (robot position dist: map exitPosition) + penalty ].
g run.
map drawRobotPath: (robot followOrders: g result).
map open
```

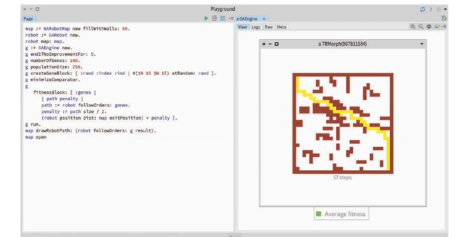


***Figure 11-3.*** *Short robot footprint*



CHAPTER 12

# Building Zoomorphic Creatures

Genetic algorithms are often used to simulate aspects of how biological individuals behave. This chapter is about artificial life. It defines and creates what we call *zoomorphic creatures*. We refer to zoomorphic creatures as virtual beings that own particular traits of biological creatures. As such, a zoomorphic creature can be considered a small digital animal.

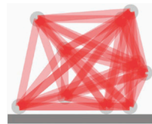


***Figure 12-1.*** *Example of a creature*

Figure 12-1 shows the example of such a creature standing on a platform. A creature is made of join points and muscles. Each muscle has two extremities and each extremity is connected to a join point. Our creatures are boneless and join points connect muscles. A join point hosts the muscle extremities.

A muscle is a complex element in our model. Each muscle oscillates and has a strength, which makes it able to resist external forces (e.g., gravity or a reaction from a platform). Muscle oscillation is regulated by an internal clock, proper to each muscle. A creature is subject to (i) gravity and (ii) the reaction force from the platform on which the creature stands. Muscles have no weight, but a join point has a weight.



© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence in Pharo*, https://doi.org/10.1007/978-1-4842-5384-7_12

CHAPTER 13

# Evolving Zoomorphic Creatures

The previous chapter presented the infrastructure that models and builds zoomorphic creatures. However, so far, the creature cannot do much: it stands where it was originally located, and we are lucky when it does not fall on its side. This chapter makes the creatures evolve to accomplish a displacement task, such as moving toward a particular direction or passing through some obstacles.

## 13.1 Interrupting a Process

Before jumping in and running the genetic algorithm, it is important to highlight an aspect of the Pharo programming language and environment.

Making creatures evolve is a very costly operation. Depending on your hardware configuration, you may have to let your computer evolve the creatures for hours. As such, most of the scripts in this chapter require a long time to complete. You should be familiar with the way that Pharo can be interrupted by pressing the Cmd and . (period) keys on MacOSX. On Windows or Linux, you use the Alt and . keys.

Interrupting Pharo opens up a Pharo debugger. When this happens, the execution has been interrupted. You may then do either of the following:

- Evaluate the code (e.g., to accurately monitor the computation progresses), which would happen in the debugger itself or in the playground
- Simply resume the computation by clicking Proceed

Closing a debugger will end the ongoing computation. Keeping the debugger open means you can always resume the execution you interrupted by clicking Proceed.



© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence in Pharo*, https://doi.org/10.1007/978-1-4842-5384-7_13

Part 1: Neural network

+

Part 2: Genetic algorithm

=

Part 3: Neuroevolution

Part 1: Neural network

+

Part 2: Genetic algorithm

=

Part 3: Neuroevolution

# Deep learning

- Greedy in properly formed training data
- E.g., distinguishing a 🐱 from a 🐶 requires 2000 pictures
- Not really the way humans learn

# Neuroevolution

- Evolution of neural network is called neuroevolution
- In practice: an AI algorithm evolves another AI algorithm

CHAPTER 14

# Neuroevolution

This chapter covers the third and last part of the book. The book started with the topic of neural networks, which are computational metaphors for the biological brain. Subsequently, the book covered genetic algorithms, computational simulations of species evolution. After these two parts, the question that may naturally be asked is: Is it possible to evolve neural networks in a fashion similar to how our biological brains went through evolution over thousands of years? The answer is yes, and this evolution mechanism is the topic of this third and last part of the book. *Neuroevolution* is a form of artificial intelligence that combines neural networks and genetic algorithms.

After giving some theoretical background on different learning mechanisms, this chapter explores a simple neuroevolution mechanism, called *NeuroGenetic*.

## 14.1 Supervised, Unsupervised Learning, and Reinforcement Learning

When we discussed how a neural network operates, we learned that a neural network requires examples. In order for a neural network to learn classification patterns in a dataset (as with the Iris dataset), the dataset has to be labeled for the neural network to identify those patterns. In the case of the Iris dataset, each flower description accompanied the name of the flower. We referred to the flower name as the label of an example. Learning from a dataset that contains labels is called *supervised learning*: the machine learning algorithm learns patterns from labeled data. Supervised learning is characterized by operating on labeled data.

In many situations, obtaining a labeled dataset is not problematic. For example, Facebook has a large dataset of labeled pictures. Each time you label a friend in a picture, you provide an example that Facebook can use to improve its models. Supervised learning finds patterns in datasets for which we have the right answer, the label.



© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence in Pharo*, https://doi.org/10.1007/978-1-4842-5384-7_14

CHAPTER 15

# Neuroevolution with NEAT

NEAT is an algorithm that builds neural networks following an incremental and evolutionary process. It uses a genetic algorithm to evolve networks. In the very early generations, neural networks are very simple, composed of a few nodes and connections. However, complexity is added in each generation. NEAT supports a number of mutations, and these mutations may add new nodes or new connections. As such, networks can only become more complex over time.

NEAT was proposed in 2002 by Kenneth O. Stanley and Risto Miikkulainen in their article titled, "Evolving Neural Networks Through Augmenting Topologies," published by MIT Press. Readers who wish to know more about the design decisions of the algorithm are welcome to read the article. The article is accessible, and it can be easily found on the web.

This chapter focuses on the implementation of the NEAT algorithm. NEAT builds neural networks made of nodes and connections. This chapter is self-contained. All the code provided in this chapter is meant to be kept in a package called NEAT and each class is prefixed with the two letters, NE.

Note that we slightly simplify the original NEAT algorithm to keep the chapter size under control. In particular, we use a simplified strategy to create species and evaluate similarities between individuals.

This chapter begins with some theoretical background before diving into the NEAT implementation.

## 15.1 Vocabulary

This chapter is about using a genetic algorithm to evolve neural networks. Although we have detailed these two concepts in previous chapters, the NEAT algorithm, as originally formulated by Kenneth and Risto in 2002, comes with its own terminology.



© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence in Pharo*, https://doi.org/10.1007/978-1-4842-5384-7_15

CHAPTER 16

# The MiniMario Video Game

This chapter builds a small video game inspired by Nintendo's Mario Bros. Our version of the game is a simplification of the real Mario Bros game. The purpose of this chapter is to provide a solid and realistic base on which we can build an intelligent artificial player. The goal of this chapter is *not* to provide a wonderful gaming experience. Instead, the game is about providing a challenging scenario for exercising the NEAT algorithm covered in the previous chapter. Our game, which we call *MiniMario*, has the following characteristics:

- The game has one hero, *Mario*, located at the center of the screen.
- Mario can be controlled by using the keyboard or by an artificial player.
- Mario can move left, right, and jump.
- The map is composed of bricks and tubes, which Mario cannot go through.
- The map is populated by monsters and Mario must avoid them or the game ends.
- Monsters walk in one direction until they bump into a brick or a tube, in which case, the walking direction changes to the opposite.
- The goal of the game is to bring Mario to the right-most location of the map.

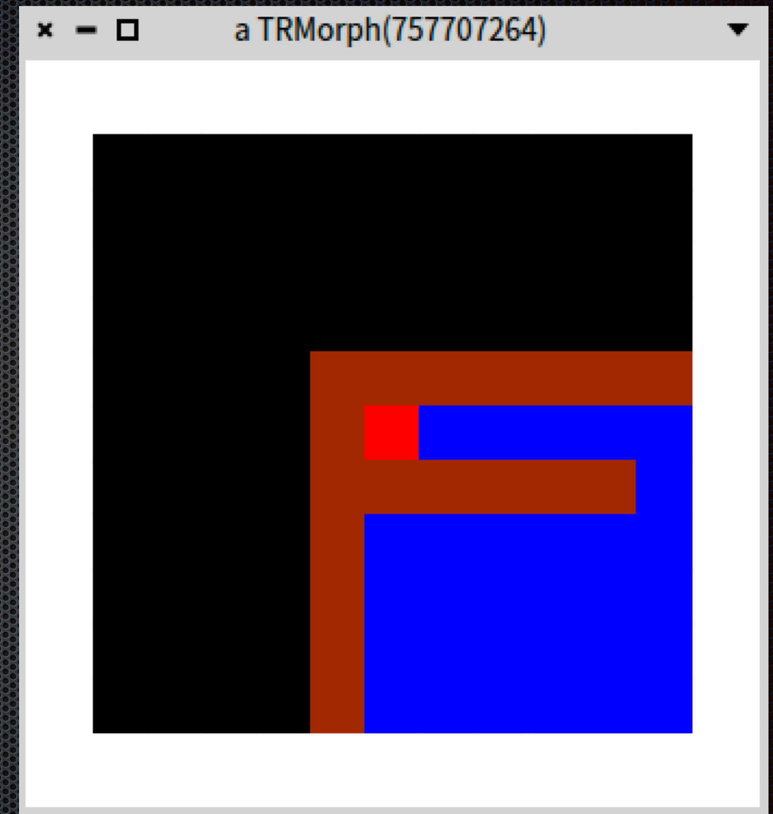
This game is driven by a global pulse, which we call a *beat*. A beat represents an indivisible time unit. At each beat, Mario and the monster may move by one cell. Note that for the sake of simplicity, a monster cannot jump.

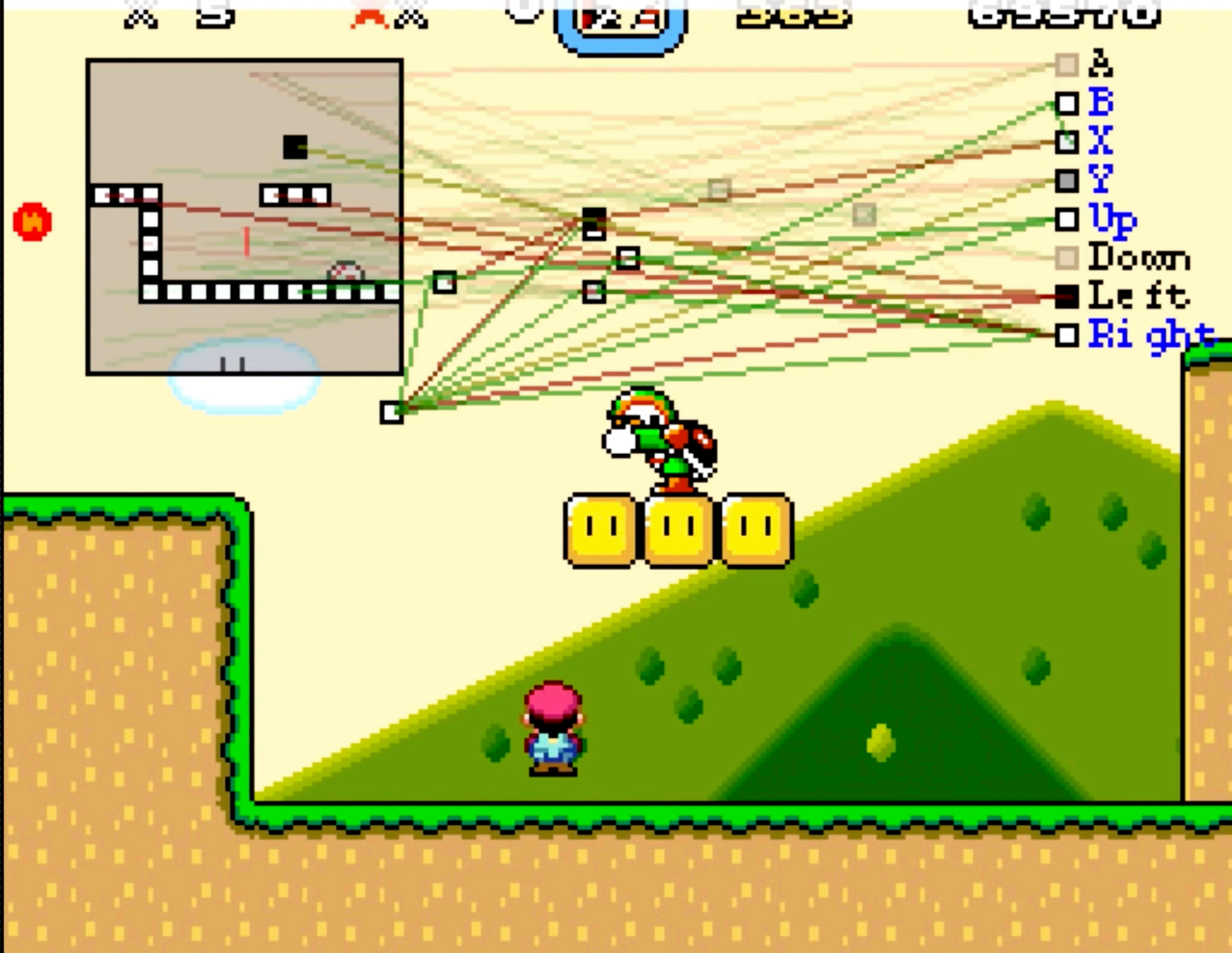


© Alexandre Bergel 2020
A. Bergel, *Agile Artificial Intelligence in Pharo*, https://doi.org/10.1007/978-1-4842-5384-7_16

```
MNWorld new seed: 7; open

MNWorld new showCompleteMap
```




Gen 34 species 14 genome 14 (37%)
Fitness: 2951 Max Fitness: 4322
A
B
X
Y
Up
Down
Left
Right

```
neat := NEAT new.
neat numberOfInputs: 121.
neat numberOfOutputs: 3.
neat populationSize: 200.
neat fitness: [ :ind |
  w := MNWorld new.
  w mario: (MNAIMario new network: ind).
  450 timesRepeat: [ w beat ].
  w mario position x ].
neat numberOfGenerations: 160.
neat run.
```

```
w := MNWorld new.
w mario: (MNAIMario new network: neat result).
w open
```

# All the code is available for free

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

# What the book is not about

- Complex deep learning models
- Transformers
- Language models
- Image recognition

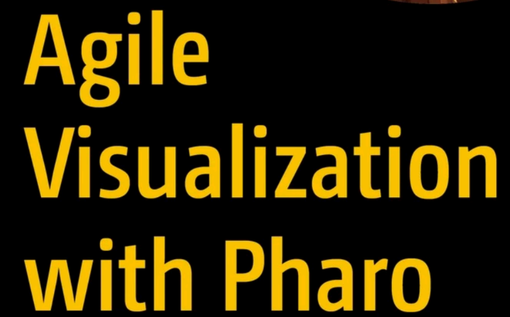
# What the book is about

- Make you an expert in Genetic Algorithms
- Introduce you to the fascinating world of Neuroevolution
- Give non-trivial and appealing examples
- Does not require a strong mathematical background

# Future plan

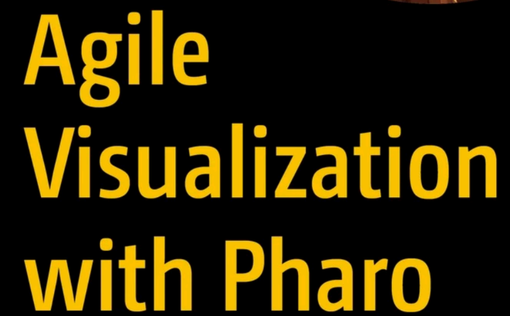
- Update book with Roassal3
- Extract the Neural Network chapters into a separate book (maybe)
- More examples about zoomorphic creatures
- Go deeper into Neuroevolution

Agile Artificial Intelligence in Pharo
Implementing Neural Networks, Genetic Algorithms, and Neuroevolution
—
Alexandre Bergel
Apress®

Agile Visualization with Pharo
Crafting Interactive Visual Support Using Roassal
—
Alexandre Bergel
Apress®

Agile Artificial Intelligence in Pharo
Implementing Neural Networks, Genetic Algorithms, and Neuroevolution
—
Alexandre Bergel
Apress



Agile Visualization with Pharo
Crafting Interactive Visual Support Using Roassal
—
Alexandre Bergel
Apress