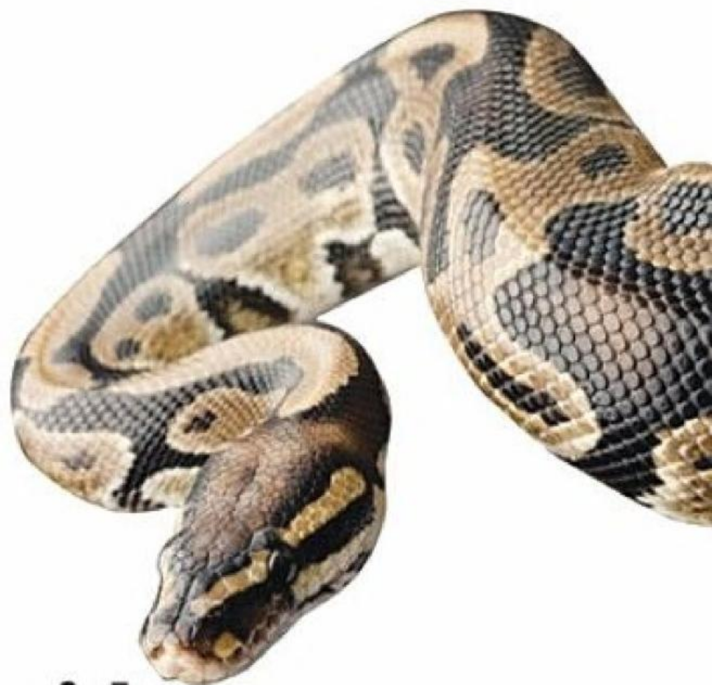


Natural Computing with PYTHON

Learn to implement genetic and evolutionary algorithms to solve problems in a pythonic way

GIANCARLO ZACCONE

Copyrighted material

Table of Contents

1. **Neural Networks**
    - Introduction
    - Structure
    - Perceptron
        - *Developing logic gates by perceptron*
    - Activation functions
        - *Linear and non-linear models*
            - *Step function*
            - *Sigmoid function*
            - *ReLU function*
        - *Sigmoid neuron*
    - How neural networks learn
    - Neural network architecture
    - Supervised learning
        - *Gradient descent*
    - MLP Python implementation
        - *Feedforward step*
        - *Backpropagation*
    - TensorFlow
    - Installation
        - *Flow graph*
        - *Placeholders*

Copyrighted material

Logistic regression
MNIST dataset
Flow graph definition
Training
Evaluation
Conclusion
Sitography
Python
Neural networks
Machine learning
TensorFlow
2. Deep Learning
Structure
What is deep learning?
Keras deep learning framework
Keras tutorial
Convolutional Neural Networks (CNNs)
Convolution layers
Pooling layers
ReLU layers
Fully connected layers
Upsampling layers
Loss layers
CNN implementation
Recurrent Neural Networks (RNNs)
Long Short-Term Memory (LSTM)
Sentiment Analysis for IMDB movie review

Copyrighted material

Autoencoders
Why copy input to output?
Use of autoencoders
Developing autoencoders
Reinforcement learning
Application areas
Elements of reinforcement learning
Q-learning
Solving the CartPole problem
Conclusion
Sitography
Deep learning
CNNs
RNNs
Autoencoders
Reinforcement learning
Keras
3. Genetic Algorithms and Programming
Structure
Evolution and algorithms
Optimization problems
Basic terminology
Genetic algorithms
Population
Fitness
Genetic operators
Python implementation
Copyrighted material

Travelling salesman problem (TSP)
Genetic programming
Terminal set and function set
Genetic operations
Symbolic regression problem using gplearn .110
Conclusion
Sitography
Genetic algorithms
Genetic programming
Python frameworks
4. Swarm Intelligence
Introduction
Structure
Mechanisms underlying collective behavior
Pheromones
Stigmergy
Stigmergy and collective behaviour
Ant colony optimisation (ACO)
ACO implementation
Particle swarm optimization
PSO implementation
SwarmPackagePy framework
Requirements
Installation
Artificial Bee Algorithm
Method invocation
Example

Copyrighted material

Conclusion
Sitography
Swarm intelligence
TSP problem
Particle swarm optimization
Ant Colony Optimization
SwarmPackagePy frameworks
5. Cellular Automata
Introduction
Structure
Background history
Automata
Turing machines
Cellular automata
Sierpiński triangle
Game of Life
Langton's ant
Wolfram's cellular automata
Implementation
CellPyLib
Rule 110
Reversibility and entropy
Sitography
Cellular automata
Turing machines
Game of Life
Langton's ant

Copyrighted material

Wolfram automata

6. Fractals

Introduction

Structure

What are fractals?

Self-similarity

Fine structure

Fractional dimensions

Recursion

Python and recursion

Fractal dimension

Cantor set

Sierpinski's fractals

Complex numbers

Python and complex numbers

Mandelbrot set

Fractals and nature

LS-Systems

Conclusion

Sitography

Fractals

Mandelbrot

Fractals and nature

7. Quantum Computing

Introduction

Structure

Quantum computers

Copyrighted material

Qubits
Quantum gates
Quantum programming
Qiskit
Programming workflow
Building a quantum circuit
Executing the quantum model
QASM backend
Quantum circuits
Quantum gates
X gate
H gate
Running Qiskit on IBM Q devices
Create a free IBM Q account to get an API token
Running on IBM Q devices
Applications of quantum computing
Conclusion
Sitography
Quantum mechanics
Quantum computing
Quantum programming
Quantum computers
Python frameworks
8. DNA Computing
Introduction
Structure
The idea behind DNA computing

Copyrighted material

DNA fundamentals
Basics of DNA computing
How to manipulate DNA
Phases of DNA algorithms
Adleman model for DNA computing
Adleman's biological approach
Python simulation of Adleman's experiment
Conclusion
Sitography
DNA computing
Adleman's experiment
Python frameworks
Index

Copyrighted material

CHEPTER 1

Neural Networks

Introduction

The history of the Neural Networks has its origins in the years the idea of neural networks learning. Between 1957 and 1958, however, *Rosenblatt* proposed the first true *modern neural network scheme*, that is, the perceptron able to recognize shapes and associate configurations.

The perceptron (described in the *Perceptron* section) exceeds the limitations of the binary structure proposed by *McCulloch* and *Pitts*, because it is equipped with variable synaptic weights, which are then able to learn.

Until the 1970s–80s, which is until the advent of modern computers, neural networks fell into the general disinterest of the scientific community, which recalls contact with the work of *Werbos* in 1974, which describes the mathematical bases for the training of multi-layer neural networks.

The most modern neural network structure is reached in 1986, when *Rumelhart*, *Hilton*, and *Williams* described the training algorithm for backpropagation of the error (introduced in *MLP Python implementation* section), thanks to which it is possible to modify the weights of the neuronal connections in a systematic way, as long as the response of the network does not become the same—or as close as possible—to the desired one.

A neural network actually presents itself as an "adaptive" system capable of modifying its structure (nodes and interconnections) based both on external data and internal information that connects and passes through the neural network during the learning phase and reasoning.

For this purpose, at the end of the chapter, the TensorFlow software library for machine learning is introduced, which provides tested and optimized modules useful in the implementation of algorithms for different types of tasks from language processing to image recognition.

Copyrighted material

Structure

- Perceptron
    - Developing logic gates by perceptron
- Activation functions
    - Linear and non-linear models
    - Sigmoid neuron
- How neural networks learn
- Neural network architecture
- Supervised learning
    - Gradient descent
- MLP Python implementation
- TensorFlow
    - Logistic regression
- Conclusion
- Sitography

> **NOTE:** The examples presented in this book were made for a Python 3.x version. However, as a matter of compatibility with all the examples that will be described in the following chapters, it is recommended to use Python 3.5.2 version, downloadable at the following link: https://www.python.org/downloads/release/python-352/

Biological neuron

Biological neurons are electrically active cells, and the human brain contains about 10^{11} neurons. They exist in different forms, although most of them have the shape shown in Figure 1.1. The main cell body is the soma; the dendrites represent the inputs of the neuron and the axon represents its output. Communication between neurons occurs at the junctions, called synapses.

Each neuron is typically connected to a thousand other neurons and, consequently, the number of synapses in the brain exceeds 10^{14}. Each neuron can be found mainly in 2 states: active or rest. When the neuron is activated, it produces an action potential (electrical

Copyrighted material

impulse) that is transported along the axon. Once the signal reaches the synapse, it causes the release of chemicals (neurotransmitters) that cross the junction and enter the body of other neurons.

Depending on the type of synapses, which can be exciters or inhibitors, these substances increase or decrease respectively the probability that the next neuron becomes active. At each synapse, a weight is associated which determines the type and extent of the exciter or inhibitor effect. Following is the diagram of a biological neuron:

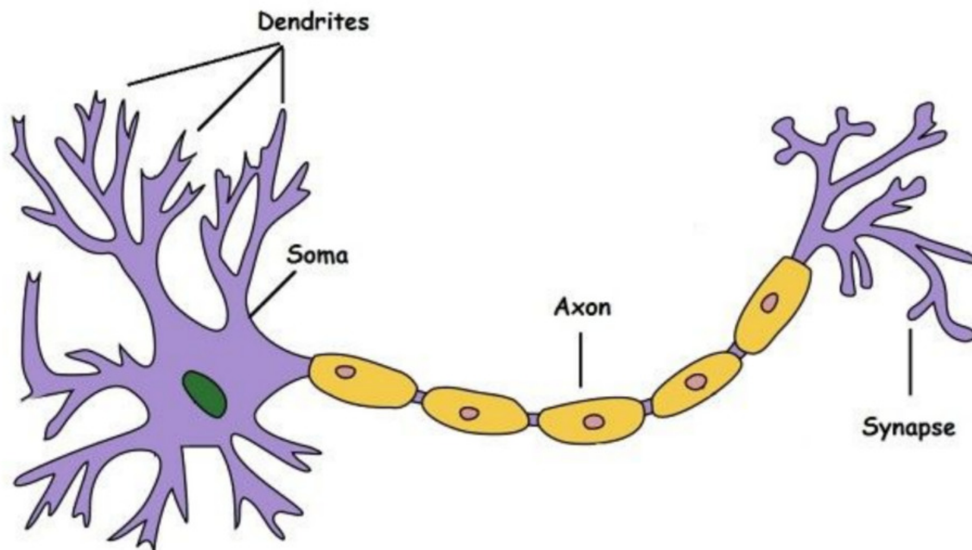


Figure 1.1 : Biological neuron representation

So, in a nutshell, each neuron makes a weighted sum of the inputs coming from the other neurons and, if this sum exceeds a certain threshold, the neuron is activated.

Each neuron, operating at a time of millisecond, represents a relatively slow processing system; however, the entire network has a very large number of neurons and synapses that can operate in parallel and simultaneously, making the actual processing power very high.

Perceptron

One type of artificial neuron is the perceptron, developed between 1950s and 60 by the scientist *Frank Rosenblatt*. A perceptron takes in input different binary values ***x1***, ***x2***, **..**, ***xn*** and produces a single binary value of output ***y***:

Copyrighted material

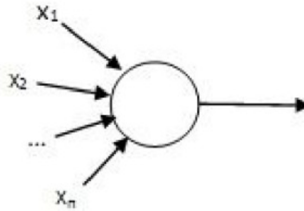


Figure 1.2 : Perceptron

Figure 1.2 shows a perceptron, which can have n input values **x1, x2, ..., xn** and a single output value **y**. Rosenblatt, in 1958, proposed simple rules for evaluating the output value, introducing the weights and matching each input value **xj** , a weight **wj**, a real value that represents the importance of input **xj** in the calculation of the value of exit **y**.

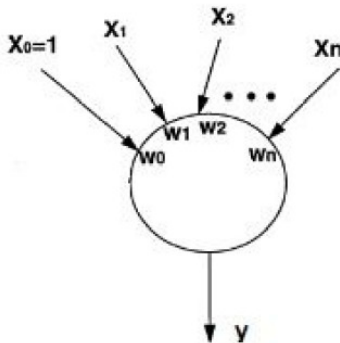


Figure 1.3 : Rosenblatt's Perceptron

The output of the perceptron is a binary value and can only assume the values 0 and 1, depending on whether the sum of the inputs weighs less than or greater than a certain threshold value, parameter of the neuron that can assume any real value:

$$output = f(x) = \begin{cases} 1 \;\; if \sum w_i x_i + b \geq threshold \\ 0 \;\; if \sum w_i x_i + b < threshold \end{cases}$$

Here, the function f (x) is the so-called **activation function**. The previous equation shows the mathematical model of a perceptron, which can also be seen as a binary decision-maker that takes its decisions by weighing inputs. If the weight of an input *xj* is greater than the others, this value will be more closely considered in the weighted sum, whereas if

Copyrighted material

the threshold value is decreased, it is more likely that the output of the neuron is *1* and vice versa.

Alternatively, to the threshold, the following relation can be considered:

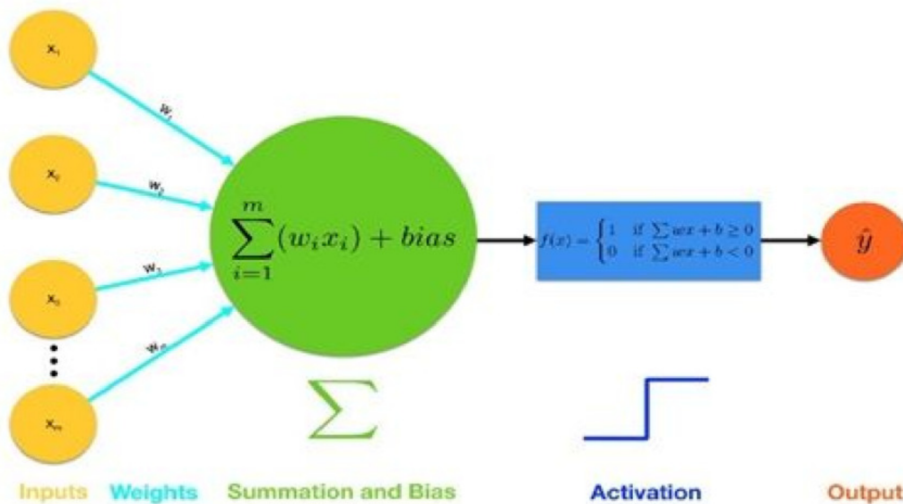
$$\sum_{i=1}^{m} w_i x_i = -bias = threshold$$

Here the activation function is

$$output = f(x) = \begin{cases} 1 & if \sum w_i x_i + b \geq 0 \\ 0 & if \sum w_i x_i + b < 0 \end{cases}$$

We can think of bias as a measure of how easy it is to get an output of the perceptron equal to *1*, it requires much less effort to activate a neuron having a very high bias and vice versa. If the bias is much less than zero then it would be very difficult to make the output of the perceptron equal to *1*. In the following, we will always use the *b* value of the bias and no longer the threshold value, this is only a small change of notation but as you will see it will make further simplifications possible.

In summary, the perceptron has the following structure:



Copyrighted material

Figure 1.4 : Complete structure of a perceptron

- **Inputs** are fed into the perceptron.
- **Weights** are multiplied to each input.
- **Summation** of the output and then addition of **Bias**.
- **Activation** function is applied. Note that here a step function is used, but there are other more sophisticated activation functions such as sigmoid, hyperbolic tangent (tanh), rectifier (relu) and so on.
- **Output** is either triggered as 1, or not, as 0. Note we use $\hat{y}$ hat to label output produced by our perceptron model.

Developing logic gates by perceptron

So far, a method has been described to evaluate the results in order to make decisions, while another way in which the earners can be used is to calculate elementary logic functions, such as AND, OR, or NAND.

For example, suppose you have a perceptron with two inputs, each with weight **1** and one bias equal to **-2** as shown in the following figure:

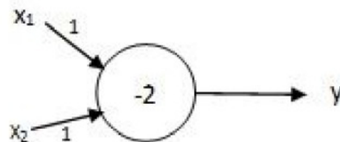


Figure 1.5 : Neural network AND gate

If you try to set the different values for *x1* and *x2* and analyze the value of the output *y*:

$$x_1 = 0, x_2 = 0 \rightarrow 0 * (1) + 0 * (1) + (-2) < 0 \rightarrow y = 0$$

$$x_1 = 1, x_2 = 0 \rightarrow 1 * (1) + 0 * (1) + (-2) < 0 \rightarrow y = 0$$

$$x_1 = 0, x_2 = 1 \rightarrow 0 * (1) + 1 * (1) + (-2) < 0 \rightarrow y = 0$$

$$x_1 = 1, x_2 = 1 \rightarrow 1 * (1) + 1 * (1) + (-2) \geq 0 \rightarrow y = 1$$

It can be seen that the perceptron behaves exactly like a logical AND gate.

Copyrighted material

If we had used a value of pair 2 for both *w1* and *w2* and a bias of 3, we would have implemented a NAND logic port and in fact, thanks to De Morgan's laws related to the Boolean logic, we could use networks of perceptrons to compute any logical function.

The computational universality of the perceptrons could be reassuring and disappointing at the same time; it may be reassuring that this neuron looks as powerful as any computational device, but it may be disappointing to think that the perceptrons are nothing but a new type of NAND gate, not to mention the fact that we should design and set the weights and deviations manually, which in case of large networks would be an expensive process. Fortunately, there are the so-called *learning algorithms* that allow the automatic adjustment of weights and deviations of a network of artificial neurons.

These algorithms allow the use of artificial neurons in a completely different way from the logic gates. A neural network can simply learn to solve the problem, sometimes too complex to solve it through the explicit design of a network of logical gates.

Figure 1.6 shows a perceptron where ***x1***, ***x2*** are input signals, ***y*** is an output signal, ***w0*** is a bias, and ***w1***, ***w2*** are weights:

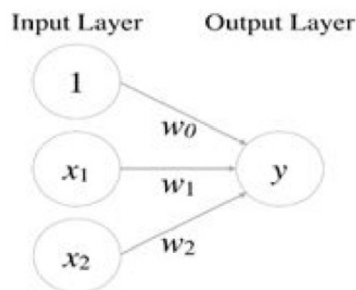


Figure 1.6 : Logical gate neural network

The output is 1, only if the sum of inputs is over thresholds. In this case, the activation function is represented as follows:

$$y = \begin{cases} 0\ (w_0 + w_1x_1 + w_2x_2) \leq 0 \\ 1(w_0 + w_1x_1 + w_2x_2) > 0 \end{cases}$$

Let us build a logic gate with this function. If *w0=-1.5*, *w1=1*, and *w2=1*, it will be the AND gate.

This is the truth table for AND, OR, and NAND:

Copyrighted material

AND			OR			NAND		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	1	0	1
0	1	0	0	1	0	0	1	1
1	1	1	1	1	1	1	1	0

Table 1.1 : AND OR NAND truth tables

Let us implement it with the following Python code:

1. First, let us import the `numpy` library:

```
import numpy as np
```

2. Define the AND logic gate:

```
def And(x1, x2):
 x = np.array([1, x1, x2])
 w = np.array([-1.5, 1, 1])
 y = np.sum(w*x)
 if y <= 0:
 return 0
else:
 return 1
```

3. Define the OR logic gate:

```
def Or(x1, x2):
 x = np.array([1, x1, x2])
 w = np.array([-0.5, 1, 1])
 y = np.sum(w*x)
 if y <= 0:
 return 0
else:
 return 1
```

4. Define the NAND logic gate:

Copyrighted material

```
def Nand(x1, x2):
 x = np.array([1, x1, x2])
 w = np.array([1.5, -1, -1])
 y = np.sum(w*x)
 if y <= 0:
 return 0
 else:
 return 1
```

5. Now, we build the `main` function:

```
if __name__ == '__main__':
```

6. Now, define the input array:

```
 input = [(0, 0), (1, 0), (0, 1), (1, 1)]
```

7. Let us start the evaluation of the logic gates:

```
 print("AND")
 for x in input:
   y = And(x[0], x[1])
   print(str(x) + " -> " + str(y))

 print("OR")
 for x in input:
   y = Or(x[0], x[1])
   print(str(x) + " -> " + str(y))

 print("NAND")
 for x in input:
   y = Nand(x[0], x[1])
   print(str(x) + " -> " + str(y))
```

8. Here are the results:

```
AND (0, 0) -> 0
(1, 0) -> 0
(0, 1) -> 0
(1, 1) -> 1
```

Copyrighted material

```
OR
(0, 0) -> 0
(1, 0) -> 1
(0, 1) -> 1
(1, 1) -> 1
NAND
(0, 0) -> 1
(1, 0) -> 1
(0, 1) -> 1
(1, 1) -> 0
```

Activation functions

In biological neurons, the action potential is transmitted in full once the potential difference to the membranes exceeds a certain threshold. In a certain sense, this is also true for artificial neurons, in particular to an artificial neuron. It calculates a linear function of its inputs and applies a non-linear function, called the **activation function**, to the result.

What would happen if the activation function were not applied? The neural network as a whole would be the composition of linear functions, which in turn is a linear function. In other words, a neural network of 1000 linear layers would have the same representative capacity as a network with a single linear layer. The activation function therefore serves to break the linearity and make the network a non-linear function.

In the next section, we will explain the difference between linear and non-linear models, and how the use of these models (or simply functions) is crucial for the implementation of artificial neural networks.

Linear and non-linear models

A neural network without an activation function is simply a regression model, that is, it tries to access data distribution with a straight line (see Figure 1.7). In this example, we can see that the line represents the distribution in a rather imprecise way. The level would always be the same as the previous one:

Copyrighted material

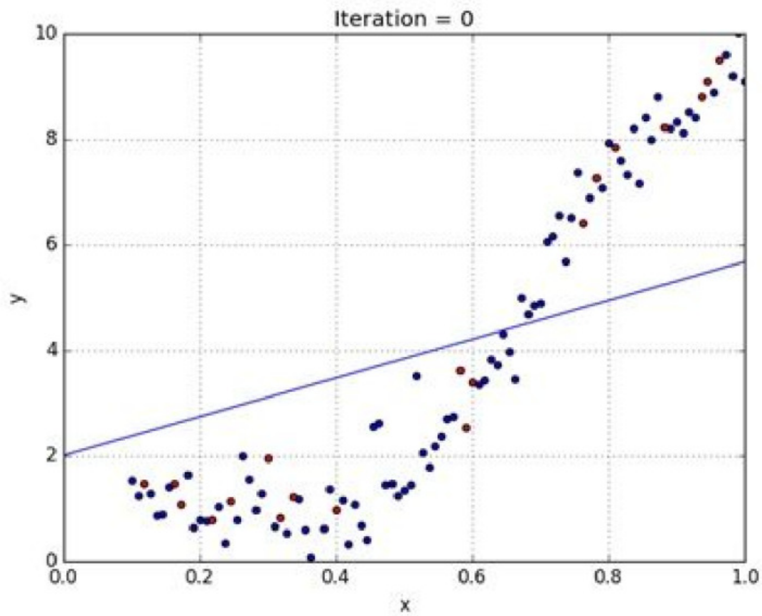


Figure 1.7 : Linear regression

The purpose of neural networks is to be a universal function approximator, that is, to be able to approximate any function, and to do this, it is necessary to introduce a factor of non-linearity, here which is the introduction of the activation functions.

Copyrighted material

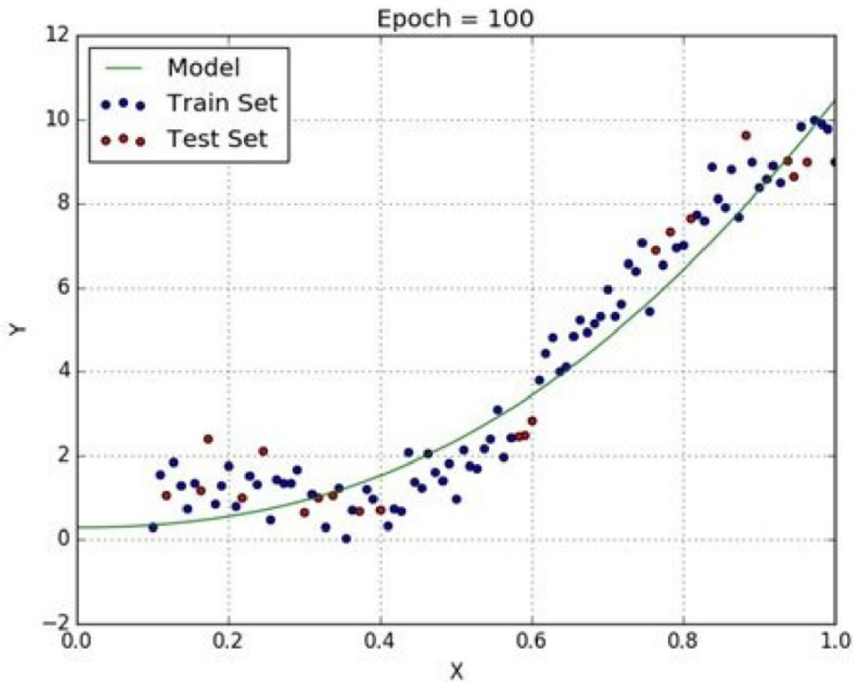


Figure 1.8 : Non-linear regression

In the preceding graph, with a non-linear model it is possible to approximate the same data much more precisely. Thus, to be usable, an activation function must be non-linear.

The three most used activation functions are the step function, the sigmoid, and the ReLU function, which is described in the following sections.

Step function

The most intuitive function is the step function, in a sense, more similar to biological functioning. For all negative values, the answer remains 0, while it jumps to +1 as soon as the value reaches or even exceeds zero. The advantage is that it is simple to calculate, and it normalizes the output values by compressing them all in a range between 0 and +1.

However, this type of function is not really used because it is not differentiable at the point where it changes value (that is, there are no derivatives at that point). The derivative is nothing more than the slope of the tangent line at that point, and it is fundamental in deep learning, as it determines the direction in which to orientate for adjustments to values.

In short, we can say that this abrupt change of state makes it difficult to control the

Copyrighted material

behavior of the network. A small change on a weight could improve the behavior for a given input but make it jump completely for other similar ones.

Sigmoid function

Sigmoid function is the most used activation function It has similarities with the step function, but the transition from 0 to +1 is more gradual, with an s-shaped trend. The advantage of this function, besides being differentiable, is to compress the values in a range between 0 and 1 and therefore be very stable even for large variations in values. The sigmoid has been used for a long time, but it still has its problems.

It is a function that has a very slow convergence, given that for very large input values the curve is almost flat, with the consequence that the derivative tends to zero. This poor responsiveness toward the ends of the curve tends to cause problems of vanishing gradient, which we will discuss later. Also not being zero-centered, the values in each learning step can only be all positive or all negative, which slows down the training process of the network.

This is a function that is no longer widely used in the intermediate layers, but still very valid in output for categorization tasks.

ReLU function

The **ReLU** (**rectifier linear unit**) function is a function that has become very popular lately, especially in intermediate layers. The reason is that it is a very simple function to calculate: it flattens the response to all negative values to zero, while leaving everything unchanged for values equal to or greater than zero.

This simplicity, combined with drastically reducing the problem of vanishing gradient, makes it a particularly attractive feature in intermediate layers, where the number of steps and calculations is important. In fact, calculating the derivative is very simple: for all the negative values it is equal to zero, while for the positive ones it is equal to 1. At the angular point in the origin instead the derivative is indefinite but is nevertheless set to zero by convention.

We can plot them, by the following Python code:

```
import numpy as np

import matplotlib.pylab as plt
```

Copyrighted material

```
def step(x):
   return np.array(x > 0, dtype=np.int)

def sigmoid(x):
   return 1 / (1 + np.exp(-x))

def relu(x):
   return np.maximum(0, x)

x = np.arange(-5.0, 5.0, 0.1)
y_step = step(x)
y_sigmoid = sigmoid(x)
y_relu = relu(x)
plt.plot(x,    y_step,    label='Step',    color='k',    lw=1,
linestyle=None)
plt.plot(x, y_sigmoid, label='Sigmoid', color='k', lw=1, ls='-
-')
plt.plot(x,    y_relu,    label='ReLU',    color='k',    lw=1,
linestyle='-.')
plt.ylim(-0.1, 1.1) plt.legend()
```

Here, is the output for the three functions, which is coded above:

Copyrighted material

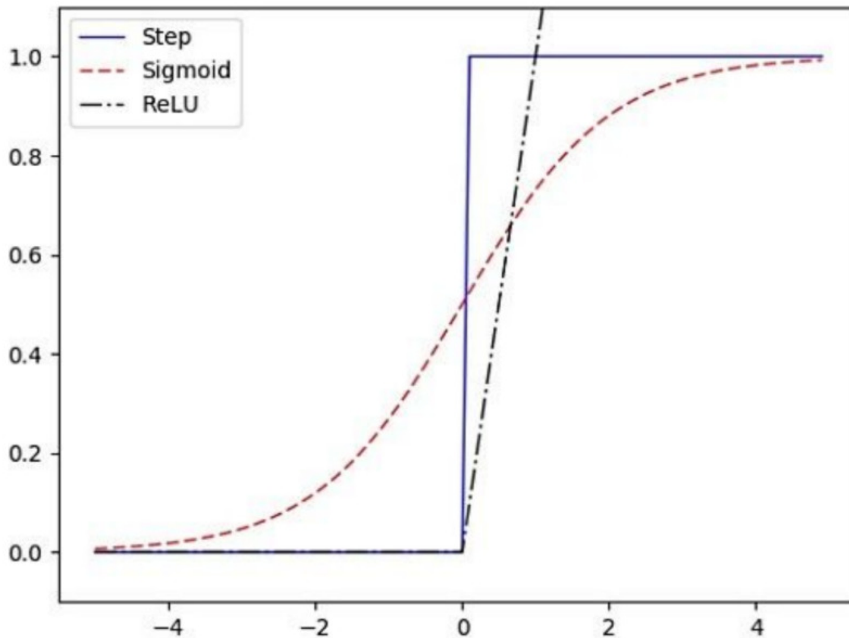


Figure 1.9 : ReLu, step, and sigmoid function

Sigmoid neuron

The sigmoidal neuron is represented in the same way as the perceptron, it can have an input *x1*, *x2*, ..., *xn*, the corresponding weights *w1*, *w2*, ...,*wn*, the deviation *b*, and the output *y*, which can be replicated and applied as an input to other neurons in a network.

Unlike the case of the perceptron, the *xi* inputs of the sigmoidal neuron can take any real value between 0 and 1, as well as its output *y*, which is defined by:

$$y = \sigma(w \cdot x + b)$$

Where represents the sigmoid function:

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

Sigmoid neurons have the useful characteristic that their input and output values can assume any value between 0 and 1, which allows better modeling of more realistic systems, such as in the case of taking binary decisions as in the case of the percone, it could implement the convention to interpret any output greater than or equal to *0.5* as a true

Copyrighted material

Boolean result and any input value less than *0.5* as a false result.

Here we can visualize the sigmoid with its derivative:

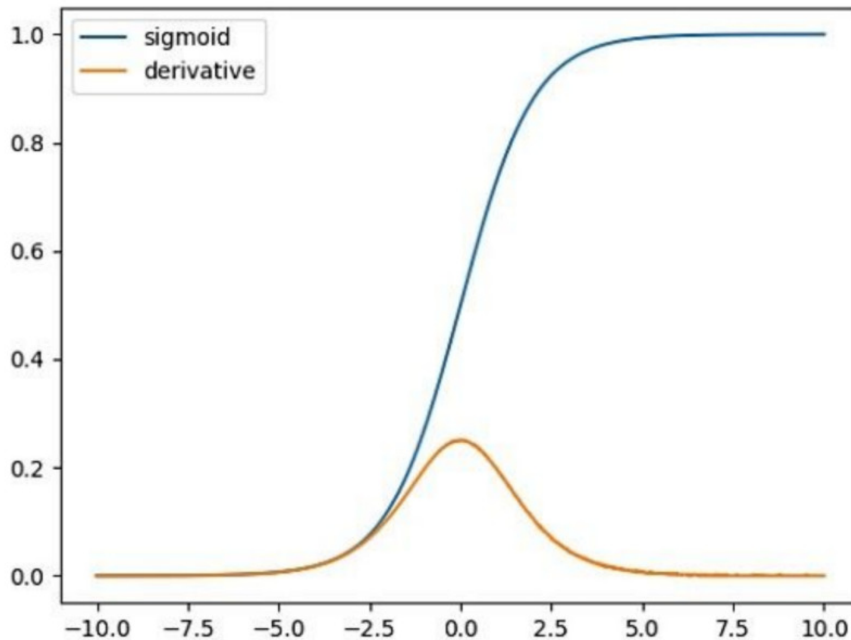


Figure 1.10 : Sigmoid function and its derivative

Following is the python code for the same:

```
import numpy as np
import matplotlib.pyplot as plt


def sigmoid(x):
   return 1 / (1 + np.exp(-x))

def derivative(x, step):
   return (sigmoid(x+step) - sigmoid(x)) / step
x = np.linspace(-10, 10, 1000)

y1 = sigmoid(x)
```

Copyrighted material

```
y2 = derivative(x, 0.0000000000001)

plt.plot(x, y1, label='sigmoid')

plt.plot(x, y2, label='derivative')

plt.legend(loc='upper left')

plt.show()
```

How neural networks learn

The main purpose of learning a neural network is to set all the weights and deviations so as to obtain a network that makes small changes in the value of the network correspond to small changes in weights (or deviations). If it is true that small changes in weights (or deviations) correspond to small changes in the output, then this capacity could be exploited to ensure that the network behaves in the most appropriate way depending on the situation. For example, if you have a network for the classification of images that misjudges the image of a dog as cat class, a small change in the weights and deviations of the network could cause the network to approach the correct classification, in this situation it is said that the network is learning. Unfortunately, this is not what happens when the artificial neural network contains perceptrons, in fact a small change in weights or deviations in a single perceptron could cause that percussion output to change, for example, from 1 to 0, this change could cause a big change in network behavior. So even if the network correctly classifies the image of the cat, it is very likely that for every other image behavior has changed drastically and uncontrollably. This is essentially the reason for the success of a type of artificial neuron called sigmoidal, similar to the perceptron but suitable for obtaining small changes in output at small changes in weights or deviations.

Neural network architecture

As already mentioned in the previous paragraph, neural networks are models of machine learning that try to imitate the structure and functioning of the biological brain, consisting of large clusters of neurons linked together by axons, through the use of sets of neural units, called artificial neurons, interconnected to form a network. Each neural unit is connected with many others, and the link can be of a reinforcing or inhibitory nature with respect to the activation of the units to which it is connected. Each neuron contains a function used to combine the values of all its inputs and a function, called an activation function that returns the output of the neuron. The general form of the overall function contained in a neuron is represented by the following formula: $y=f(\Sigma wixi + bi)$

Copyrighted material

In this,*wi* are the weights assigned to each input in the combination phase and *b* a term, called bias, which is added later. The set of weights and bias represents the information that the neuron learns during the training phase and which it subsequently retains. The function *f* represents the activation function, which normally consists of a threshold or limitation function that makes sure that only signals with values compatible with the threshold, or the imposed limit, can propagate to the next neuron or neurons. Typically, the activation function is a nonlinear function, and usually it is a step function, a sigmoid, or a logistic function.

Neural networks are typically structured into three parts, containing distinct amounts of neurons:

- An input level: input layer.
- A more or less numerous set of hidden internal levels: hidden layer.
- An output level: output layer.

The input signals pass through the entire network from the input level to the output level, passing from the neurons of the inner layers, as illustrated in the following figure:

Copyrighted material

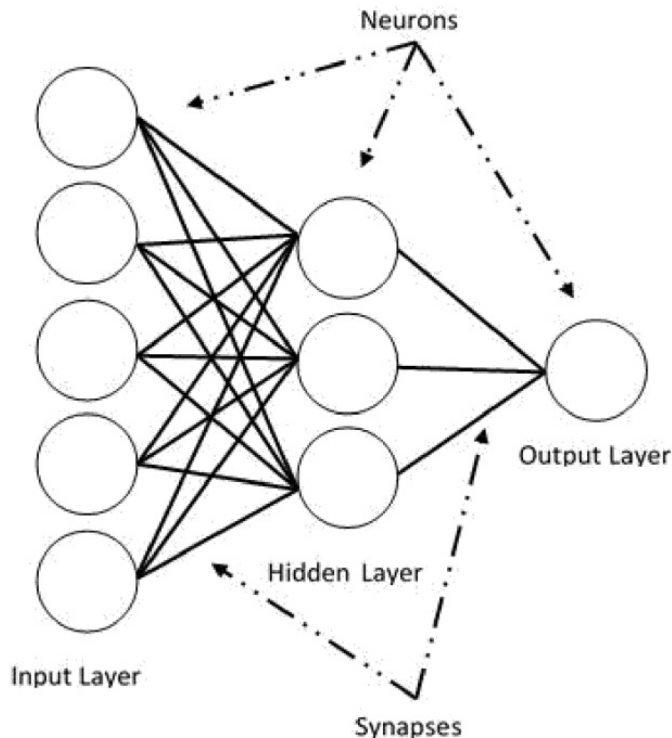


Figure 1.11 : Sample structure of a neural network with two hidden levels

These networks with more than one hidden layer are called multi-layer-perceptron (MLP) (this name, however, is improper, since MLPs are not made of percons, but sigmoid neurons).

Neural networks of this type, in which the output of a layer is used as input for the next layer, are called feedforward networks (or even **forward networks**): this means that the neurons of the same level cannot communicate between them, neither with those of lower level, and in the network there are no cycles.

The input–output link, or the network transfer function, is obtained from an unscheduled learning process, based on empirical data.

These techniques are part of **m**achine learning (**ML**): it is an application of **artificial intelligence** (**AI**) that provides systems that are able to automatically learn and improve from experience without being explicitly programmed. Machine learning focuses on the development of computer programs.

Copyrighted material

The process can be as follows:

- Supervised (supervised learning), if you have a set of training data, including typical examples of inputs with their corresponding outputs. The goal is to predict the output value for each valid input value, based only on a limited number of matching examples (pairs of input–output values). The network is trained through a suitable algorithm (**backpropagation**); if successful, learn to recognize the unknown relation that links the input variables to the output variables, and can therefore make predictions even if the output is not known a priori.
- Unsupervised (unsupervised learning), it is based on training algorithms referred to a set of data that includes only the input variables. These algorithms group the input data and identify suitable clusters to represent them.
- Reinforcement (reinforcement learning), in which a suitable algorithm identifies a certain modus operandi starting from a process of observation of the external environment; every action has an impact on the environment, and the environment produces a feedback that drives the algorithm itself in the learning process.

Supervising learning

In the case of supervised learning, the training phase of a network consists in estimating the weights content contained in each neuron that minimize the error between the expected output values from the training data and the predicted values from the network. The function that calculates this error can be of different types and is called loss **function** or cost **function**.

The estimation of the weights can be obtained with mathematical optimization techniques such as gradient descent by backpropagation.

The Supervised Learning algorithm is based on a two-step cyclical technique: forward-propagation and updating of weights by backpropagation.

In the first phase of forward-propagation, the inputs pass through the entire network from the input level to the output level; the outputs produced are then recovered and through the loss function the prediction error is calculated with respect to the expected outputs.

This error is used to calculate the gradient of the loss function, which is then propagated backward into the network (hence the term backpropagation) until each neuron gets its

Copyrighted material

gradient value. At this point, the weight update phase begins: the calculated values of the gradient are fed to **gradient descend** algorithm, which uses them to update the weights of each neuron, with the intent of minimizing the loss function value.

Gradient descent

Gradient descent is a technique that aims to minimize the cost function as much as possible. Imagining the cost function as a function of only two variables (to simplify), the purpose of our gradient descent is to find the global minimum of the function, which is the lowest point.

In this simplified case, the minimum seems obvious enough, but in most cases the functions are much more complex, and you have to get there by successive approximations. Trying to simplify at most with an analogy, all that the gradient descent does is start from a random point, and then based on the derivatives move in one direction or another. A high derivative means high slope, therefore still far from the minimum, and the next step will be wide. A small derivative means slight slope, therefore close to the minimum, which involves smaller approach steps.

MLP Python implementation

We see in practice how to implement an MLP for learning the XOR function expressed in the following table:

x0	*x1*	*y*
0	0	0
0	1	1
1	0	1
1	1	1

Table 1.3: XOR truth table

The network to be implemented is shown in the diagrammatic representation:

Copyrighted material

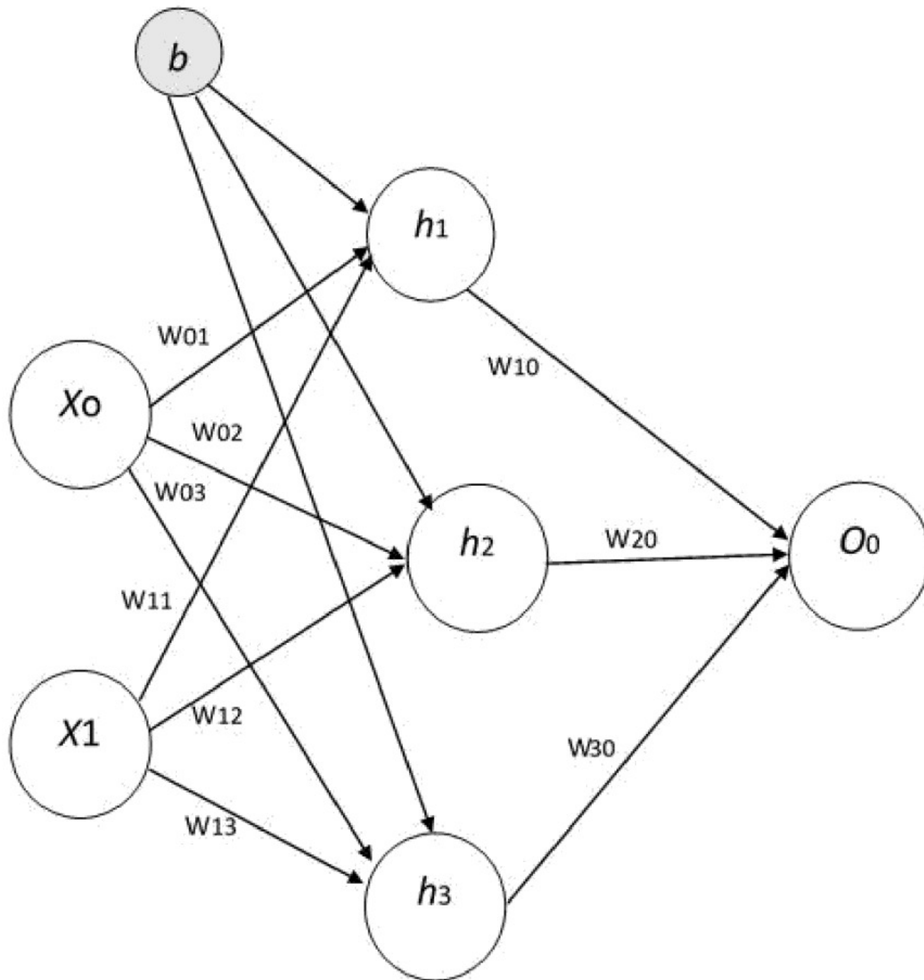


Figure 1.12 : MLP neural network for XOR problem

In the previous diagram, you can see an MLP with one input layer, one hidden layer, and one output layer.

- The input layer represents the data set, each sample has three features (***x0***, ***x1***).
- The hidden layer consists of five neurons (***h1***, ***h2***, ***h3***).
- The output layer consists of one neuron (o_0).

We start to set up the Python code for our problem:

```
import numpy as np

class NeuralNetworkXOR:
```

Copyrighted material

```
def __init__(self, numHidden, numPasses, learningRate):
    self.inpDim = 2
    self.outDim = 1
    self.numHidden = numHidden
    self.numPasses = numPasses
    self.learningRate = learningRate
    self.x = np.array([[0,0],[0,1],[1,0],[1,1]], np.float64)
    self.y = np.array([[0],[1],[1],[0]], np.float64)
```

Where `numHidden` is 3, `numPasses` is 25000, `learningRate` is 1.

First of all, all bias weights and values are initially valued randomly, which means that in the first pass the network response will be random and most likely completely incorrect:

```
def initializeWeights(self):
    self.w1 = np.random.rand(self.inpDim, self.numHidden)
    self.b = np.random.uniform(-0.5, 0.5, size = self.numHidden)
    self.w2 = np.random.rand(self.numHidden, self.outDim)
    self.b0 = np.random.uniform(-0.5, 0.5, size = self.outDim)
```

The first step is to calculate what is called the **cost function**, which is a function that in some way represents the average quadratic error (as the difference between output and expected value) of all outputs.

Feedforward step

The input values x_0 and x_1 are now entered into the network.

Each neuron in the hidden layer: *h1*, *h2*, *h3* gets the weighted sum of the input values as input,

$$z_j = \sum_{i=0}^{1} x_i w_{ij} + b \quad j = 1,2,3 z_j = \sum_{i=0}^{1} x_i w_{ij} + b \quad j = 1,2,3:$$

Copyrighted material

$$z_1 = x_0 w_{01} + x_1 w_{11} + b_1 \quad for\ j = 1$$
$$z_1 = x_0 w_{01} + x_1 w_{11} + b_1 \quad for\ j = 1$$

$$z_2 = x_0 w_{02} + x_1 w_{12} + b_2 \quad for\ j = 2$$
$$z_2 = x_0 w_{02} + x_1 w_{12} + b_2 \quad for\ j = 2$$

$$z_3 = x_0 w_{03} + x_1 w_{13} + b_3 \quad for\ j = 3$$
$$z_3 = x_0 w_{03} + x_1 w_{13} + b_3 \quad for\ j = 3$$

Each neuron calculates an output, a_h, by using the activation function :

$$a_h = \sigma(z_h)$$

The procedure is the same for the output layer neuron O_0. The output layer gets the weighted sum of the output values of the hidden layer

$z_0 = \sum_{i=1}^{3} a_i w_{i0} + b_0 \quad i = 1,2,3$:$z_0 = \sum_{i=1}^{3} a_i w_{i0} + b_0 \quad i = 1,2,3$:

$z_0 = a_1 w_{10} + a_2 w_{20} + a_3 w_{30} + b_0$

The output layer neuron calculates an output by using an activation function:

$$a_o = \sigma(z_0) a_o = \sigma(z_0).$$

As we have the final output for our whole data set, we can now calculate the cost function:

$$cost = \frac{1}{2}(\overline{y} - y)^2$$

Here is the code for the forward propagation:

```
def forwardPropagate(self):
    self.z1 = np.dot(self.x, self.w1) + self.b
    self.a1 = self.sigmoid(self.z1)
    self.z2 = np.dot(self.a1, self.w2) + self.b0
    self.yHat = self.sigmoid(self.z2)
    self.cost = 0.5 * np.sum(np.square(self.y - self.yHat))
```

Typical activation functions for neural networks are sigmoid, ReLU, or tanh. In our case we will use sigmoid:

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

Copyrighted material

Let us code the sigmoid function in Python using `numpy`:

```
def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))
```

Backpropagation

The backpropagation algorithm was proposed in 1986 by Rumelhart, which aims to train an MLP neural network. In this section, we will give a detailed description of this algorithm, because it constitutes the working nucleus of a neural network. The backpropagation algorithm uses the gradient descent criterion to minimize the cost function, defined as the mean square error between the desired outputs and those obtained. So, we split the computation for the output layer and the inner layer.

Let us see first how to update the weights of the output layer. We need to calculate the variation of the cost function with respect to the output weights, it means, we need to evaluate the following partial derivatives:

Copyrighted material

$$\frac{\partial E}{\partial w_{i0}} \qquad for\ i = 1,2,3$$

Applying the chain rule, we have

$$\frac{\partial E}{\partial w_{i0}} = \frac{\partial E}{\partial a_0}\frac{\partial a_0}{\partial z_0}\frac{\partial z_0}{\partial w_{i0}} \qquad for\ i = 1,2,3$$

, where:

$$\frac{\partial E}{\partial a_0} = \bar{y} - y$$

$$\frac{\partial a_0}{\partial z_0} = \frac{\partial \sigma(z_0)}{\partial z_0} = a_0\,(1 - a_0)$$

, while the last term is

$$\frac{\partial z_0}{\partial w_{i0}} \quad for\ i = 1,2,3$$

It becomes,

$$\frac{\partial z_0}{\partial w_{10}} = \frac{\partial(a_1 w_{10} + a_2 w_{20} + a_3 w_{30} + b_0)}{\partial w_{10}} = a_1 \quad for\ i = 1$$

$$\frac{\partial z_0}{\partial w_{20}} = \frac{\partial(a_1 w_{10} + a_2 w_{20} + a_3 w_{30} + b_0)}{\partial w_{20}} = a_2 \quad for\ i = 2$$

$$\frac{\partial z_0}{\partial w_{30}} = \frac{\partial(a_1 w_{10} + a_2 w_{20} + a_3 w_{30} + b_0)}{\partial w_{30}} = a_3 \quad for\ i = 3$$

So, finally, we can compute the variation of the cost function with respect to the output weights:

$$\frac{\partial E}{\partial w_{10}} = (\bar{y} - y)\,a_0\,(1 - a_0)\,a_1 \qquad for\ i = 1$$

$$\frac{\partial E}{\partial w_{20}} = (\bar{y} - y)\,a_0\,(1 - a_0)\,a_2 \qquad for\ i = 2$$

$$\frac{\partial E}{\partial w_{30}} = (\bar{y} - y)\,a_0\,(1 - a_0)\,a_3 \qquad for\ i = 3$$

Copyrighted material

Now we update the weights of the inner layer:

$$\frac{\partial E}{\partial w_{ij}} \qquad for\ i = 0{,}1\ \ and\ j = 1{,}2{,}3$$

For the sake of simplicity let us consider the case *for i=0 and j=1*

$$\frac{\partial E}{\partial w_{01}} = \frac{\partial E}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial w_{01}}$$

The first term of the equation above is computed using the chain-rule for the partial derivatives:

$$\frac{\partial E}{\partial a_1} = \frac{\partial E}{\partial a_0}\frac{\partial a_0}{\partial z_0}\frac{\partial z_0}{\partial a_1}$$

$$\frac{\partial E}{\partial a_1} = (\bar{y} - y)\, a_0\,(1 - a_0)\,\frac{\partial(a_1 w_{10} + a_2 w_{20} + a_3 w_{30} + b_0)}{\partial a_1} = (\bar{y} - y)\, a_0\,(1 - a_0)\, w_{10}$$

The second term is:

$$\frac{\partial a_1}{\partial z_1} = a_1(1 - a_1)$$

The last and third term is equal to:

$$\frac{\partial z_1}{\partial w_{01}} = \frac{\partial(x_0 w_{01} + x_1 w_{11} + b_1)}{\partial w_{01}} = x_0$$

Putting all together we can compute $\frac{\partial E}{\partial w_{01}}\frac{\partial E}{\partial w_{01}}$::

$$\frac{\partial E}{\partial w_{01}} = (\bar{y} - y)\, a_0\,(1 - a_0)\, w_{10} a_1 (1 - a_1)\, x_0$$

Generalizing the above relations, we can build the following formulas for the output layer and the input layer weights update:

- Output layer:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} = (\bar{y} - y)\, a_0\,(1 - a_0)\, a_i \qquad for\ i = 1{,}2{,}3\ and\ j = 0$$

Copyrighted material

- Input layer:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} = (\bar{y} - y)\, a_0\, (1 - a_0)\, w_{j0}\, a_j\, (1 - a_j)\, x_i \quad for\ i = 0{,}1\ and\ j\ = 1{,}2{,}3$$

Finally, we can write the Python function for the backpropagation:

```
def backPropagate(self):
     self.delta2 = np.multiply(-(self.y - self.yHat), self.
sigmoidPrime(self.z2))
     self.dcdw2 = np.dot(self.a1.T, self.delta2)
     self.delta1 = np.dot(self.delta2, self.w2.T) * self.
sigmoidPrime(self.z1)
     self.dcdw1 = np.dot(self.x.T, self.delta1)
```

To update the weights of the network, let us multiply the values for the learning rate parameter, it is assumed to be equal to 1, even if it could be changed in order to fit the learning rate of the network:

```
  def updateWeights(self):
     self.w1 -= self.dcdw1 * self.learningRate
     self.b1 -= np.mean(self.dcdw1) * self.learningRate
     self.w2 -= self.dcdw2 * self.learningRate
     self.b2 -= np.mean(self.dcdw2) * self.learningRate
```

The neural network must be trained in order to learn the function to compute:

```
  def train(self):
     self.initializeWeights()
  for i in range(self.numPasses):
     self.forwardPropagate()
     self.backPropagate()
     self.updateWeights()
     if i % 5000 == 0:
        print("Iteration {i} \nOutput:\n {y} \ \nPredicted
Output:\n {yHat} \nCost:\n{cost}
                    \n".format(i = i, y=self.y, \yHat =
```

Copyrighted material

```
self.yHat, \cost = self.cost))
```

Putting all together the above functions, we built the following `NeuralNetworkXOR` Python class, which can be easily managed:

```
import numpy as np

class NeuralNetworkXOR:
  def __init__(self, numHidden, numPasses, learningRate):
    self.inpDim = 2
    self.outDim = 1
    self.numHidden = numHidden
    self.numPasses = numPasses
    self.learningRate = learningRate
    self.x = np.array([[0,0],[0,1],[1,0],[1,1]], np.float64)
    self.y = np.array([[0],[1],[1],[0]], np.float64)

  def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

  def sigmoidPrime(self, x):
    return np.exp(-x) / ((1 + np.exp(-x)) ** 2)

  def initializeWeights(self):
    self.w1 = np.random.rand(self.inpDim, self.numHidden)
    self.b1 = np.random.uniform(-0.5, 0.5, size = self.numHidden)
    self.w2 = np.random.rand(self.numHidden, self.outDim)
    self.b2 = np.random.uniform(-0.5, 0.5, size = self.outDim)

  def forwardPropagate(self):
    self.z1 = np.dot(self.x, self.w1) + self.b1
    self.a1 = self.sigmoid(self.z1)
    self.z2 = np.dot(self.a1, self.w2) + self.b2
    self.yHat = self.sigmoid(self.z2)
```

Copyrighted material

```
        self.cost = 0.5 * np.sum(np.square(self.y - self.yHat))

        def backPropagate(self):
        self.delta2 = np.multiply(-(self.y - self.yHat),\
                      self.sigmoidPrime(self.z2))
        self.dcdw2 = np.dot(self.a1.T, self.delta2)
        self.delta1 = np.dot(self.delta2, self.w2.T) *\
        self.sigmoidPrime(self.z1)
        self.dcdw1 = np.dot(self.x.T, self.delta1)

    def updateWeights(self):
        self.w1 -= self.dcdw1 * self.learningRate
        self.b1 -= np.mean(self.dcdw1) * self.learningRate
        self.w2 -= self.dcdw2 * self.learningRate
        self.b2 -= np.mean(self.dcdw2) * self.learningRate

    def train(self):
        self.initializeWeights()
        for i in range(self.numPasses):
            self.forwardPropagate()
            self.backPropagate()
            self.updateWeights()
            if i % 5000 == 0:
                print("Iteration {i} \nOutput:\n {y} \
                    \nPredicted Output:\n {yHat}
                    \nCost:\n{cost} \n"
                   .format(i = i, y=self.y, \
                                   yHat = self.yHat, \
                                   cost = self.cost))


def main():
    nn = NeuralNetworkXOR(numHidden = 3,\
```

Copyrighted material

```
            numPasses = 25000,\
            learningRate = 1)
    nn.train()


if __name__ == '__main__':
    main()
```

Executing the preceding code, we should have a result like this:

```
Iteration 0
Output:
 [[ 0.]
 [ 1.]
 [ 1.]
 [ 0.]]
Predicted Output:
 [[ 0.7912961 ]
 [ 0.82359783]
 [ 0.85018254]
 [ 0.86919411]]
Cost:
0.7176054575188995

Iteration 5000
Output:
[[ 0.]
[ 1.]
[ 1.]
[ 0.]]
Predicted Output:
```

Copyrighted material

```
[[ 0.02727036]
[ 0.97273709]
[ 0.97378694]
[ 0.02004276]]
Cost:
0.0012878877387594275

Iteration 10000
Output:
[[ 0.]
[ 1.]
[ 1.]
[ 0.]]
Predicted Output:
[[ 0.01876351]
[ 0.98165778]
[ 0.98236969]
[ 0.01285129]]
Cost:
0.0005822448937529495

Iteration 15000
Output:
[[ 0.]
[ 1.]
[ 1.]
[ 0.]]
Predicted Output:
[[ 0.01521737]
```

Copyrighted material

```
[ 0.98528372]
[ 0.9858567 ]
[ 0.01004788]]
Cost:
0.0003745650823309048

Iteration 20000
Output:
[[ 0.]
[ 1.]
[ 1.]
[ 0.]]
Predicted Output:
[[ 0.01314728]
[ 0.98737285]
[ 0.98786532]

[ 0.00847111]]
Cost:
0.00027565309400540334
```

As you can note, after 20000 steps the cost function is almost zero and the predicted output is very similar to the expected result.

TensorFlow

This section presents the main concepts of the TensorFlow framework and we will also see a first concrete case through the implementation of a logistic regression algorithm for a problem of image classification.

Installation

To install TensorFlow (release 1.12) compatible with Python 3.5.2 distribution, I suggest

Copyrighted material

the following procedure:

1. Go to the following link:

https://www.tensorflow.org/install/pip#package-location

Figure 1.13 : TensorFlow package location

2. Depending on the OS used and the Python version, copy the relative URL. For example, for Windows OS and Python 3.5.2, you should use the following link:

https://storage.googleapis.com/tensorflow/windows/gpu/tensorflow_gpu-1.12.0-cp35-cp35m-win_amd64.whl

Then paste the link on the address bar of your web browser to start the download of the TensorFlow package.

3. Open the windows command prompt then `cd` to where you've downloaded the `tensorflow_gpu-1.12.0-cp35-cp35m-win_amd64` file and use the `pip` installer:

```
pip tensorflow_gpu-1.12.0-cp35-cp35m-win_amd64
```

Copyrighted material

Flow graph

TensorFlow is a library designed for numerical computation through *flow graph.* A flow graph is a direct acyclic graph that represents a computation. In TensorFlow, each node is an operation (called *ops*) that takes zero or more tensors into input and returns zero or more tensors. Defining a computation using a flow graph is natural, as shown in the following diagram:

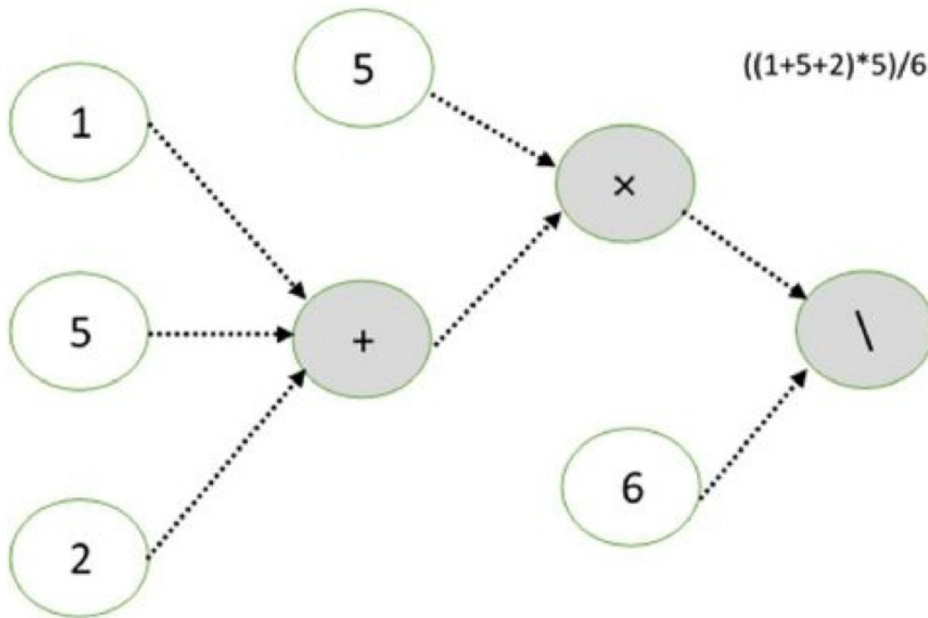


Figure 1.14 : TensorFlow—flow graph

- Green nodes represent constants, a constant is an operation that takes zero tensors into input and returns a tensor (for example);
- Node + takes in input three tensors (**1, 5, 2**) and returns a tensor (**1 + 5 + 2**);
- Nodes **x** and **/** take **2** tensors in input and one remains.

Now let us define this simple graph in TensorFlow:

```
import tensorflow as tf

a = tf.constant(1.)

b = tf.constant(5.)

c = tf.constant(2.)

d = tf.constant(5.)
```

Copyrighted material

```
e = tf.constant(6.)
result = ((a+b+c)*d)/6
```

Once the flow graph is defined, to calculate the result we must first launch a session of the current graph:

```
sess = tf.Session()
```

At this point we ask TensorFlow to return the value of the `result` tensor:

```
r = sess.run(result)
print (r)


6.66667
```

Clearly, it is possible to break the computation and give a name to the intermediate tensors:

```
x = a+b+c
y = x*d
result = y/6


print (sess.run(x))
print (sess.run(y))
print (sess.run(result))


8.0
40.0
6.66667
```

When we have finished with the graph, we can close the session. This will destroy the content of the tensors we have defined.

Placeholders

We have seen how to define constants in TensorFlow. The constants are designed to

Copyrighted material

maintain the same value at each execution of the graph. If we want to be able to define the input values used in the flow graph at runtime, we must create placeholders (placeholders) as follows:

```
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
c = tf.placeholder(tf.float32)
d = tf.placeholder(tf.float32)
e = tf.placeholder(tf.float32)
result = ((a+b+c)*d)/6
```

When we create a placeholder, we must not define its value, as it must be defined in the computation phase (and not in the definition phase of the graph). We must, however, define the type of data of the placeholder (a `float` in our case).

At this point, we can create the session and start the computation by specifying the values of the placeholders using a dictionary as follows:

```
sess = tf.Session()
print sess.run(result,feed_dict={a:1,b:5,c:2,d:5,e:6})
sess.close()
```

```
6.66667
```

Placeholders and constants make it possible to define the input values of the flow graph. We have also seen that it is possible to give names to the tensors in output (for example, `result`). When we train a model, we need structures that allow us to maintain and update parameters. To achieve this level of persistence, the variables are used. We define a variable as follows:

```
var = tf.Variable(0)
```

When we create a variable, we must specify how to initialize it. In this case, the variable will be initialized with a zero. Before you can access the variable, it must be explicitly initialized. To do this, we need to create an `initializer` node:

```
init = tf.global_variables_initializer()
```

Copyrighted material

We can then create the session, execute the initialization, and read the value of `var`:

```
sess = tf.Session()

sess.run(init)

print sess.run(var)
```

```
0
```

Now let us create a node `inc` which increases the value of `var` of a unit:

```
inc = tf.assign(var,var+1)
```

The node `inc` takes two tensors into input, assigns the value of the second to the first, and returns the assigned value:

```
print(sess.run(inc))
```

```
1
```

At subsequent executions of node `inc`, the value of `var` will be incremented:

```
for i in range(3):
   print sess.run(inc)
```

This is the `result`:

```
2

3

4
```

Logistic regression

Regression is a statistical model that aims to identify the type of relationship between different factors measured on a sample of individuals. For example, if we decide to measure, on a large group of people, the height and the number of shoes of each, we will notice that in general there is a proportionality between the two sizes (usually a tall person has a long foot). Regression models this kind of relationships in an analytical form.

Logistic regression is a particular model in which the variables, both the dependent and the independent, can only assume values of 0 or 1 (that is, true or false: an example can be

Copyrighted material

x = 1 the person wears glasses, *x = 0* the person does not have glasses). In this context, we usually try to study the probability that the event y (dependent variable) occurs.

In particular, once a critical probability value has been established which discriminates whether a group belongs or not, a classification can be made based on the results.

The model used for logistic regression is precisely the sigmoid function we introduced in *Sigmoid* section of this chapter. In particular, we shall use the logistic regression model to classify images of the MNIST data set, introduced below.

MNIST dataset

The **MNIST** dataset is available at the URL: http://yann.lecun.com/exdb/mnist/

The dataset contains images of handwritten digits. Each image is grayscale, has resolution 28 x 28 pixels, and belongs to one of the 10 classes: 0.1, ..., 9. The dataset is divided into three parts:

- Training, which contains 55,000 images;
- Test, which contains 10000 images;
- and Validation, which contains 5000 images.

TensorFlow provides utilities to download the dataset from the internet and load it into the memory. We start by importing the `input_data` utility:

```
from tensorflow.examples.tutorials.mnist import input_data
```

So, we load the dataset (it may take a few minutes):

```
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

Training, testing, and validation are contained in `mnist.train`, `mnist.validation`, and `mnist.test` respectively. Each subset of the data set contains the images (for example, `inst.validation.images`) and the corresponding labels (for example,

Copyrighted material

`mnist.validation.labels`). Let us start by exploring the training set:

```
print (mnist.train.images.shape)
```

```
(55000, 784)
```

The training images are contained in a 55000 x 784 matrix. In practice, each of the samples (each row of the matrix) is the list of 784 pixels that make up the corresponding image. To visualize the image, we will have to carry out a *reshape* of the vector in order to obtain a 28 x 28 matrix:

```
from matplotlib import pyplot as plt
plt.figure()
img = mnist.train.images[0].reshape((28,28))
plt.imshow(img,cmap='gray')
plt.show()
```

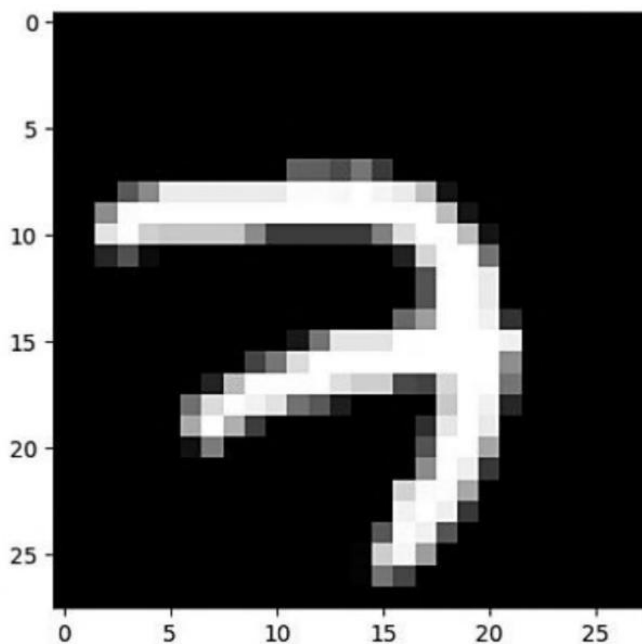


Figure 1.15 : MNIST image

Copyrighted material

The labels are instead represented with an *onehot* type encoding.

According to this coding, the number is represented by a dimension vector containing one in the hex position (the first position is the one corresponding to i = 0) and zero in all the other positions. For example:

```
0 -> 100
1 -> 010
2 -> 001
```

In our case, the vectors representing the labels will be of size *n = 10*:

```
print (mnist.train.labels.shape)
```

```
(55000, 10)
```

The label corresponding to the image shown earlier is *7*:

```
print (mnist.train.labels[0])
```

```
[ 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
```

To obtain the label from the onehot vector, simply use the `argmax` function, which returns the index at which the maximum is obtained:

```
print (mnist.train.labels[0].argmax())
```

```
7
```

Flow graph definition

We define the flow graph corresponding to a linear regressor in TensorFlow to classify the images of the MNIST dataset. We start by defining the placeholder that will contain the input data:

```
x = tf.placeholder(tf.float32, [None, 784])
```

We have defined that the input data must be of the float type. It is a matrix *[None × 784]*. The first dimension `None` indicates that the array can contain an arbitrary number of

Copyrighted material

rows. The number of columns (number of features) is `784`, which is the number of pixels in the images of the MNIST dataset. We therefore define the variables that will contain the parameters necessary for linear transformation (the parameters are initialized with zeros):

```
W = tf.Variable(tf.zeros([784, 10]))
```

```
b = tf.Variable(tf.zeros([10]))
```

We calculate the scores as follows:

```
scores = tf.matmul(x,W)+b
```

So, we apply the `softmax` function to the scores:

```
y = tf.nn.softmax(tf.matmul(x, W) + b)
```

Training

Now, we need to train the algorithm. To do this, we will use the gradient descent method. It is not necessary to implement the method, as TensorFlow makes available to the *special* nodes that allow you to define the computation related to the training in a simple way. We must first define a `loss` function to be minimized during training. We will use the *crossentropy* function defined as follows:

$$H_{\acute{y}}(y) = \sum_i \acute{y}_i \log(y_i)$$

, where $y = h_\theta(X) y = h_\theta(X)$ is the predicted probability distribution (the probability vector in output) and is the distribution of *ground truth* (that is, the training sample label *X*). To calculate the cross entropy, we must introduce a placeholder for the *ý* labels that will be used during the training phase:

```
y_ = tf.placeholder(tf.float32, [None, 10])
```

The labels are in the MNIST format: *onehot* type representations.

Therefore, each label is a row of the matrix contained within `y_`. The number of lines is also variable in this case. We can therefore define the loss cross entropy:

```
loss = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_
indices=[1]))
```

Copyrighted material

The `tf.log` function calculates the logarithm of all the elements of `y_`. The vector `y_` is therefore multiplied (element by element) with `y_`. The sum of the entire resulting matrix is then made (for this `reduction_indices = [1]`). Since `y_` and `y_` could contain more than one element, the resulting loss will be the average (calculated using `tf.reduce_mean`) of all the losses. Now, we define a node that will train the network using the gradient descent algorithm:

```
train_step =
tf.train.GradientDescentOptimizer(0.1).minimize(loss)
```

Whenever we ask TensorFlow to execute the operations defined in `train_step`, it will take care of carrying out both the *forward pass* to calculate `y_` and the *backward pass* to update the parameters based on the error made in the prediction.

We must therefore insert a node that allows us to initialize all the variables:

```
init = tf.initialize_all_variables()
```

We now launch a new session for the newly defined graph and initialize the variables:

```
sess = tf.Session()

sess.run(init)
```

Now we will train the model for *minibatch*. At each iteration, we will give the network a different subset of the data contained in the dataset. In the training phase, we will ask TensorFlow to return the value of the loss to each iteration and save these values in a `losses` list:

```
losses = list()

for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    _,l=sess.run([train_step,loss],\
        feed_dict={x: batch_xs, y_: batch_ys})
        losses.append(l)
```

We plot the progress of the iteration `loss`. If the learning is successful, the value of the `loss` should decrease as the number of iterations increases:

```
plt.figure()
```

Copyrighted material

```
plt.plot(losses)
plt.xlabel('iterations')
plt.ylabel('loss')
plt.show()
```

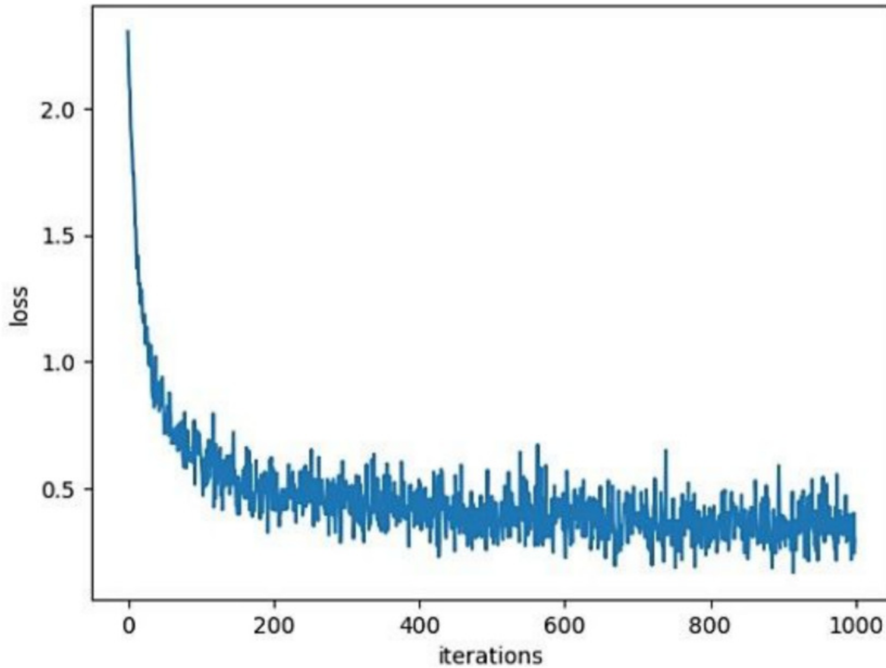


Figure 1.16 : Loss—iterations progress

Evaluation

We can now use the newly learned model to predict the labels on the whole training set:

```
predicted_probabilities=sess.run(y,feed_dict=
{x:mnist.train.images})
print predicted_probabilities.shape
print predicted_probabilities[0]
```

```
(55000, 10)
[  6.73721917e-03 3.87942465e-03 1.14390932e-01 3.13723902e-03
```

Copyrighted material

```
 1.02858699e-03 2.00961344e-03 3.94266099e-04 8.05700481e-01
 3.27259600e-02 2.99962536e-02]
```

We can transform predicted probabilities into labels using the `argmax` function:

```
predicted_labels = predicted_probabilities.argmax(1)
print predicted_labels[0]
```

```
7
```

At this point, we can evaluate the performance of the method by calculating the accuracy as done in the past. For example,

```
gt_labels = mnist.train.labels.argmax(1)
acc = (predicted_labels==gt_labels).mean()
print acc
```

```
0.901854545455
```

However, even this type of computation (evaluating performance) can be easily defined in TensorFlow. First, define a correct prediction vector:

```
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
```

This tensor will contain a list of Booleans containing `True` where the prediction is correct and `False` elsewhere:

```
print (sess.run(correct_prediction,feed_dict=\ {x:mnist.train.
images,y_:mnist.train.labels}))
[ True True True True True False True True]
```

Now, we calculate the accuracy by averaging this vector (remember that `True` is 1, while `False` is 0):

```
accuracy      =      tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))
```

Copyrighted material

We therefore obtain the accuracy on the training set as follows:

```
print                                   (sess.run(accuracy,feed_dict=\
{x:mnist.train.images,y_:mnist.
train.labels})
```

```
0.901855
```

Conclusion

In this first chapter, we introduced artificial neural networks. They are the basis of sophisticated forms of artificial intelligence, even more evolved, able to learn by exploiting mechanisms similar to those of human intelligence. Artificial neural networks are able nowadays to solve certain categories of problems, getting closer and closer to the efficiency of our brain, and even finding solutions that are inaccessible to the human mind.

A lot has been made since the birth of the concept of artificial neuron. In many and varied scientific sectors, from biomedicine to data mining, neural networks have now become a daily usage. In artificial networks, obviously, the process of machine learning is simplified compared to that of biological networks. There are no analogs of neurotransmitters, but the pattern of functioning is similar. The nodes receive input data, process it, and send information to other neurons. Through more or less numerous cycles of input-processing-output, in which the inputs present different variables, they become able to generalize and supply correct outputs associated with inputs not part of the training set.

The learning algorithms used to teach neural networks are divided into 3 categories. The choice of which to use depends on the field of application for which the network is designed and its type (feedforward or feedback). These algorithms are:

- Supervised
- Not supervised
- Reinforcement

In supervised learning, which we explored in this chapter, the network is supplied with a set of inputs to which known outputs correspond (training set). Analyzing them, the network learns the link that unites them. In this way, it learns to generalize, that is, to calculate the new correct input–output associations processing external inputs to the

Copyrighted material

sharing of a state (weights and bias) between the elements of the sequence: what is stored within the network represents a pattern that temporally links the elements of the series that the RNN analyzes.

The example above presents an output for each input, but in reality it is possible to mask part of the inputs or part of the outputs so as to obtain different combinations, some shown in the next figure; for example, you can use a many-to-one to classify a sequence of data with a single output, or use a one-to-many to label the set of subjects from an image:

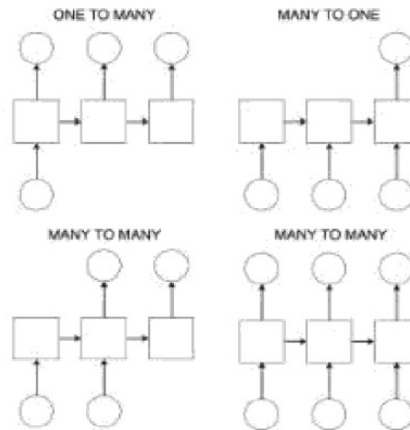


Figure 2.8 : Recombinations of RNN

Learning in the RNN takes place through the backpropagation; the difference is essentially due to the presence of common weights between the different temporal instants, so that once the unfolded structure is obtained, the modification of the single state is done respecting all the outputs: update delta is therefore computed as a sum of gradients.

The backpropagation in the RNN clearly suffers from vanishing or exploding gradients in that the unfolded structure is essentially a DNN; this implies that the RNN are not able to learn very extensive time dependencies.

Long Short-Term Memory (LSTM)

The **Long Short-Term Memory** (LSTM) represents a particular recurring network proposed by *Hochreiter* and *Schmidhuber* to solve the problem of vanishing and exploding gradient which compromised the effectiveness of the RNN. The principle behind LSTM is the memory cell, which maintains the status c outside the normal flow of the recurring network.

In fact, the state presents a direct connection with itself: the state *ct* is simply a sum between the previous state *ct-1* and the processing of the current input modulated by means of masks so that only part of them is remembered.

Copyrighted material

image
not
available