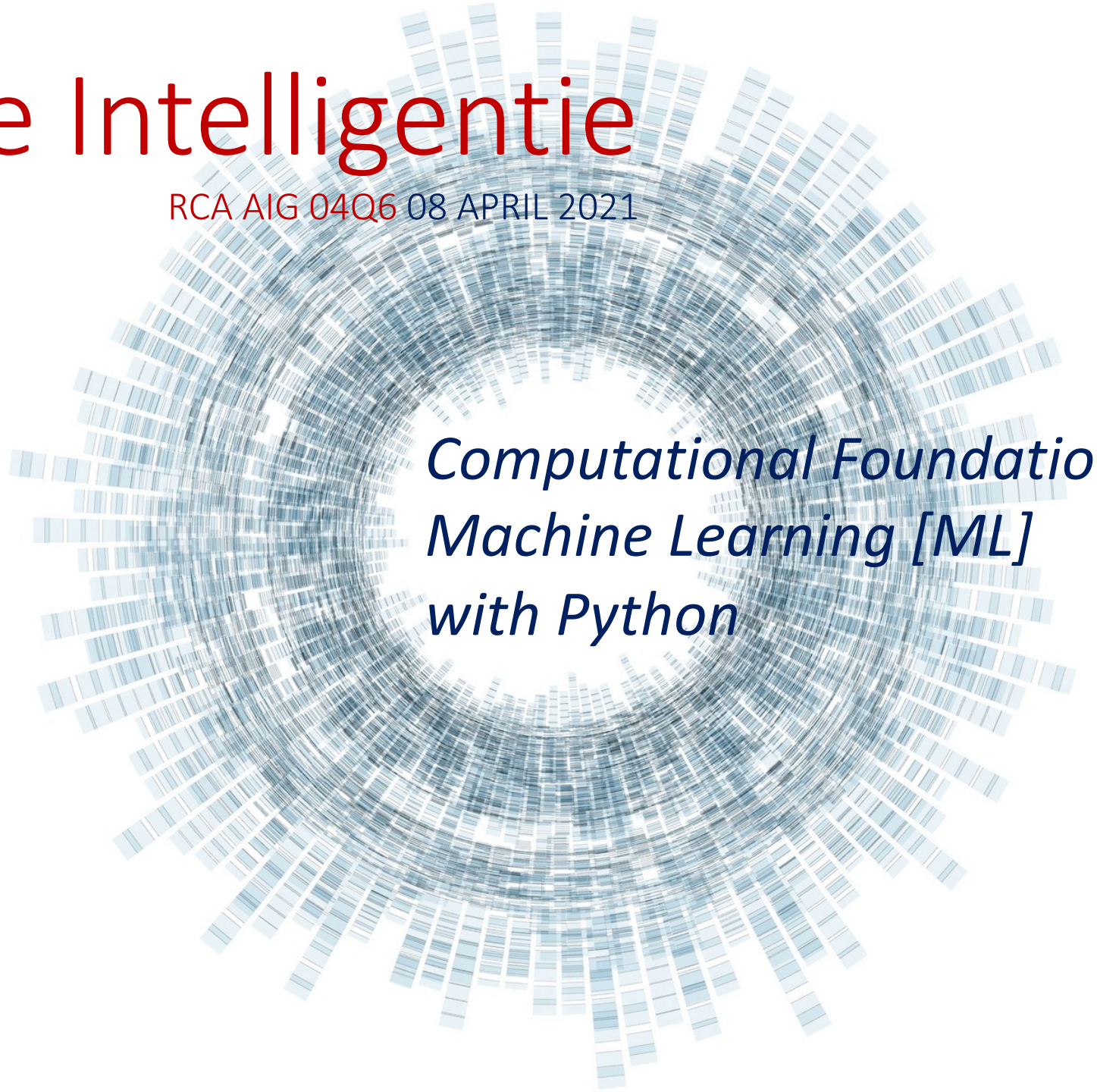
Artificiële Intelligentie

RCA AIG 04Q6 08 APRIL 2021

HOGESCHOOL ROTTERDAM



Computational Foundations of Machine Learning [ML] with Python

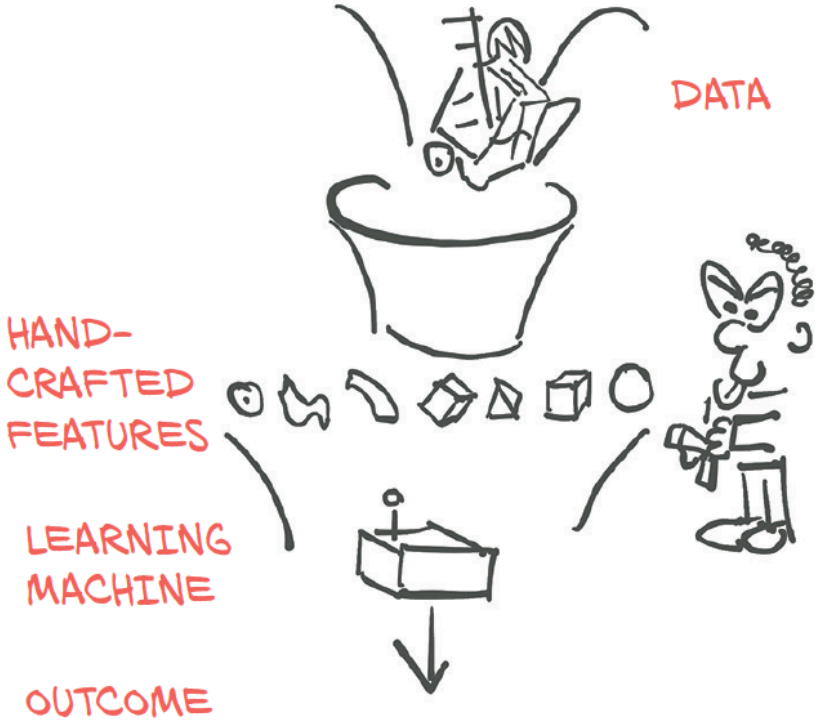
DataLab
Rob van der Willigen

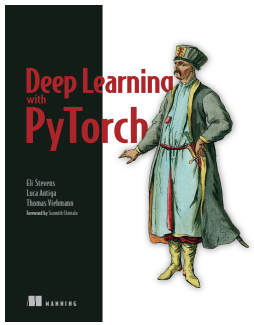
CONTEXT

- Prerequisites: basics in linear algebra, probability, and analysis of algorithms.
- Workload: homework assignments
- GitHub: Start a ML repository at GitHub

MACHINE LEARNING [ML]

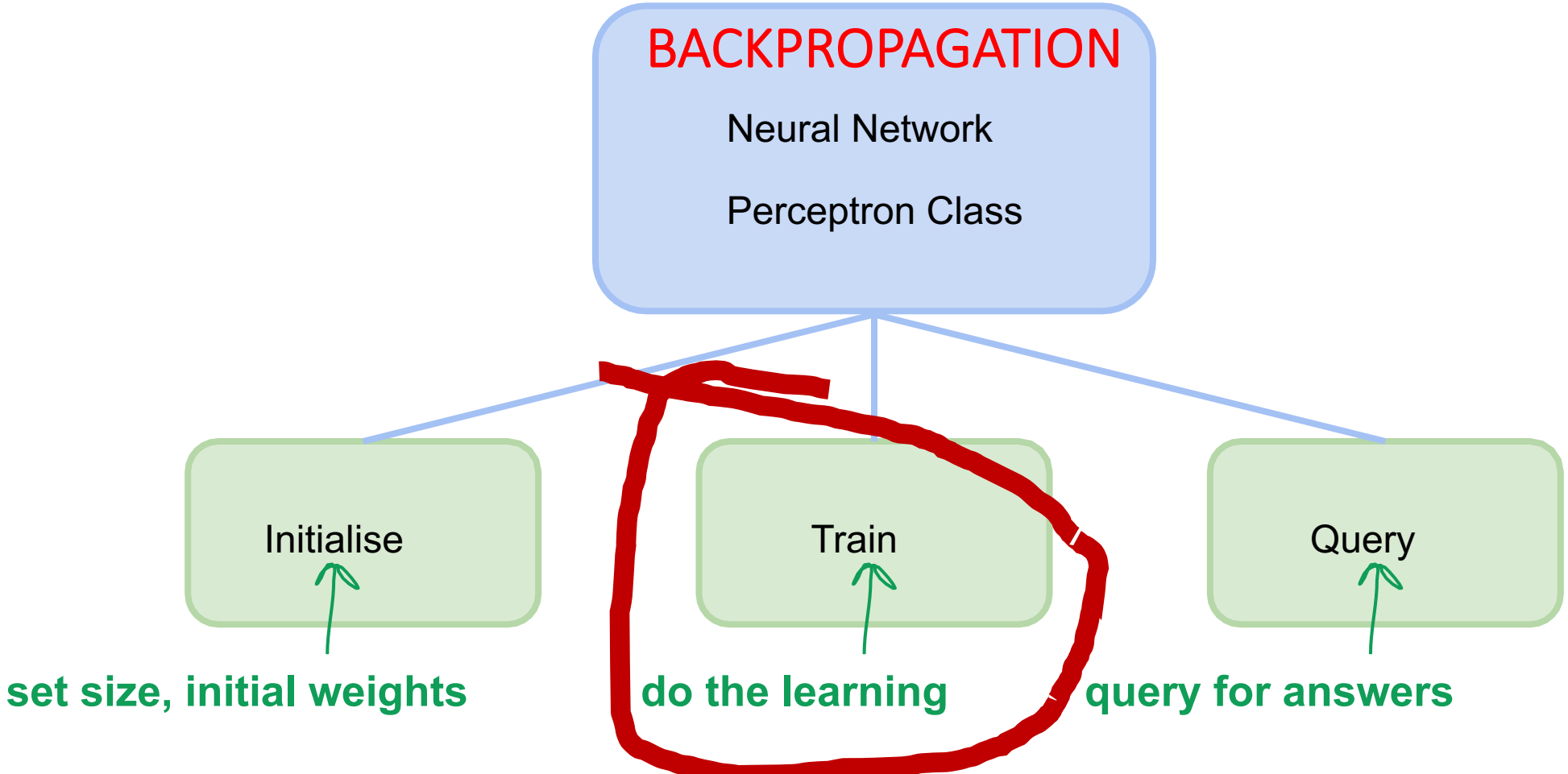
HOGESCHOOL ROTTERDAM





Lecture 04

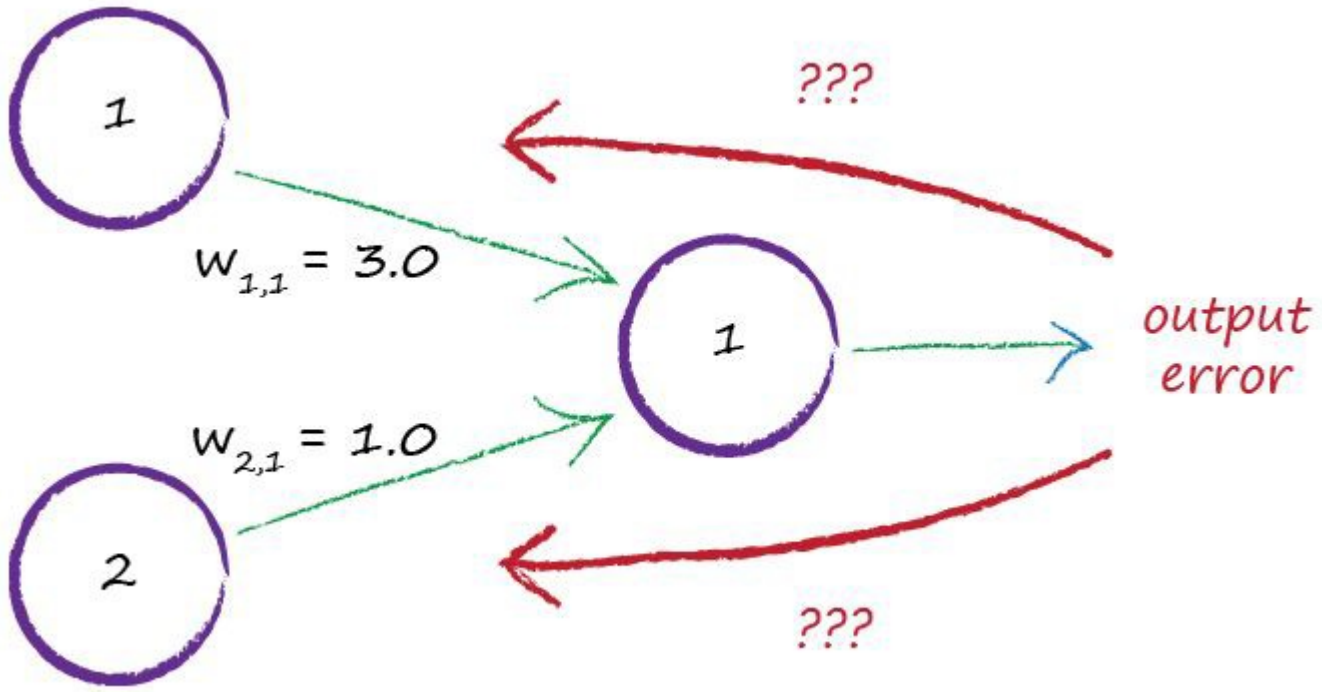
- Basic definitions and concepts of Machine Learning (ML) PART02.
- How to get from ML concepts & Models to Python code.



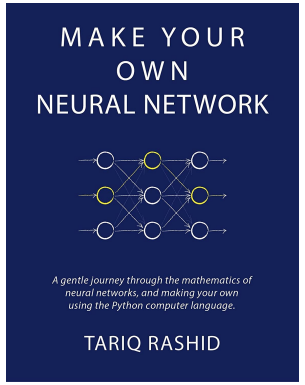
Lecture 04

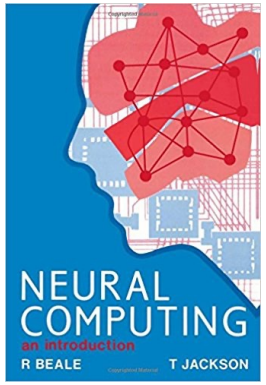
How do you get a Perceptron to learn?

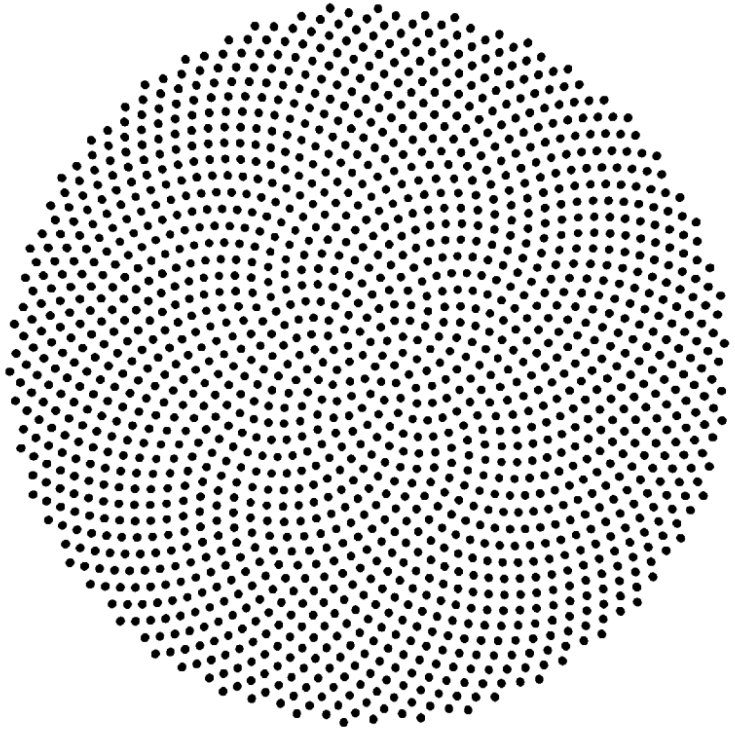
It needs to perform iterative FEED BACKWARD calculation!



{01}
Fundamentals







HOGESCHOOL
ROTTERDAM

Key-words / Concepts / Labels

Artificial neuron === Model of single biological neuron / Perceptron

Feedforward / Feed-backward / Hebbian learning

Bias / Threshold

Weights /adjustable parameters / Memory

Unit / Node / Activation Functions (Sigmoid / Step)

Input / Output / Layers / Input vs Output Space

Target / Teacher / Error / learning rate

Iterations / Epochs / Baches

Logical functions AND / OR / XOR (Boolean Algebra)

Maths + Python Key-words / Concepts

PYTHON MODULES

```
import Numpy as np
import Matlibplot.pyplot as plt
```

```
class
def return
for (loops)
if else
zip
print
list
append
+ =
```

Matrix Calculus

```
np.zeros
np.random.normal
np.dot
```

Plotting

```
plt.subplots()
plt.subplot(221)
plt.plot(epoch, error)
plt.xlabel('Epoch')
plt.ylabel('Error')
```

Perceptron

The perceptron was very promising, but it was soon discovered that is has serious limitations as it only works for linearly-separable classes.

In 1969, Marvin Minsky and Seymour Papert demonstrated that it could not learn even a simple logical function such as XOR.

This led to a significant decline in the interest in perceptron´s and Machine learning as a whole.

Perceptron

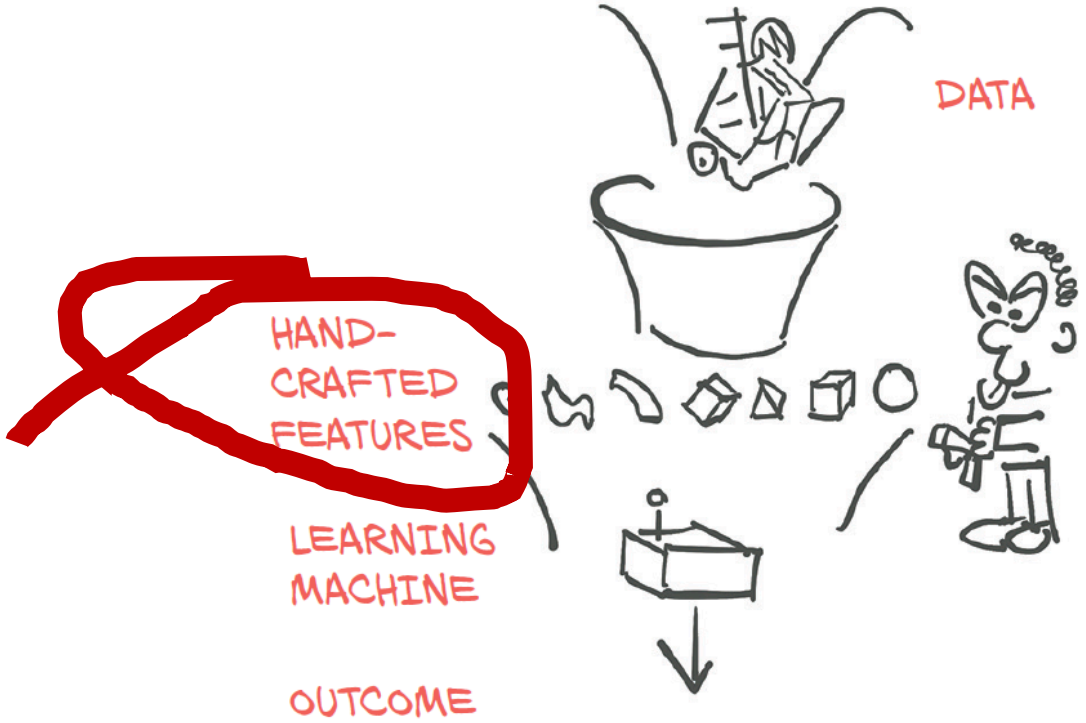
A Perceptron is a binary classifier that was invented by Frank Rosenblatt in 1958, working on a research project for Cornell Aeronautical Laboratory that was US government funded.

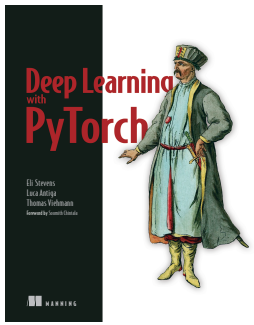
It was based on the advances with respect to mimicing the human brain, in particular the MCP architecture that was recently invented by McCulloch and Pitts.

This architecture attempted to mimic the way neurons operate in the brain: given certain inputs, they fire, and their firing behaviour can change over time.
By allowing the same to happen in an artificial neuron, researchers at the time argued, machines could become capable of approximating human intelligence.

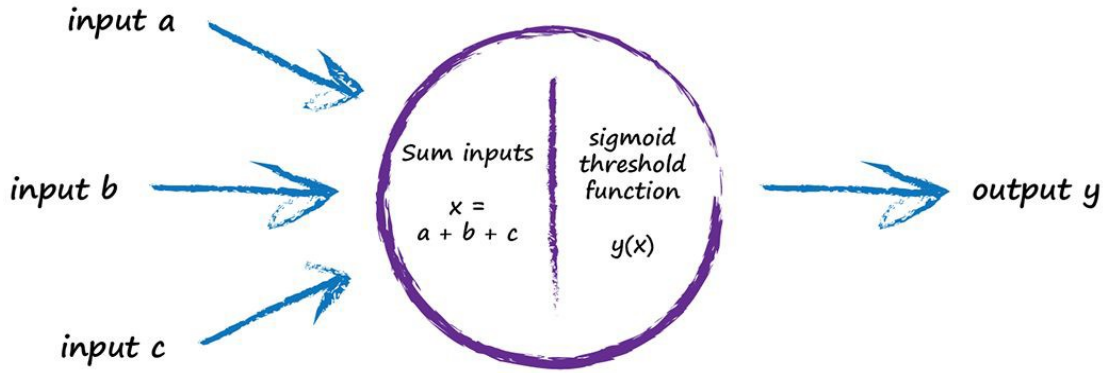
MACHINE LEARNING [ML]

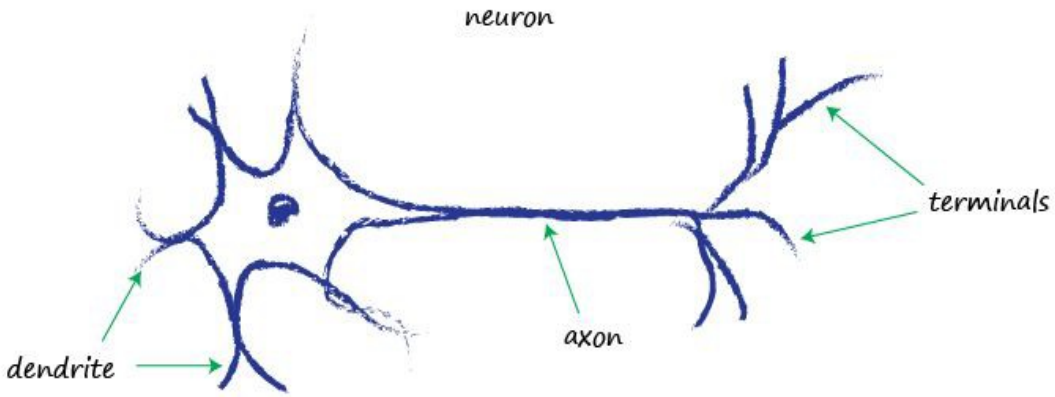
HOGESCHOOL ROTTERDAM



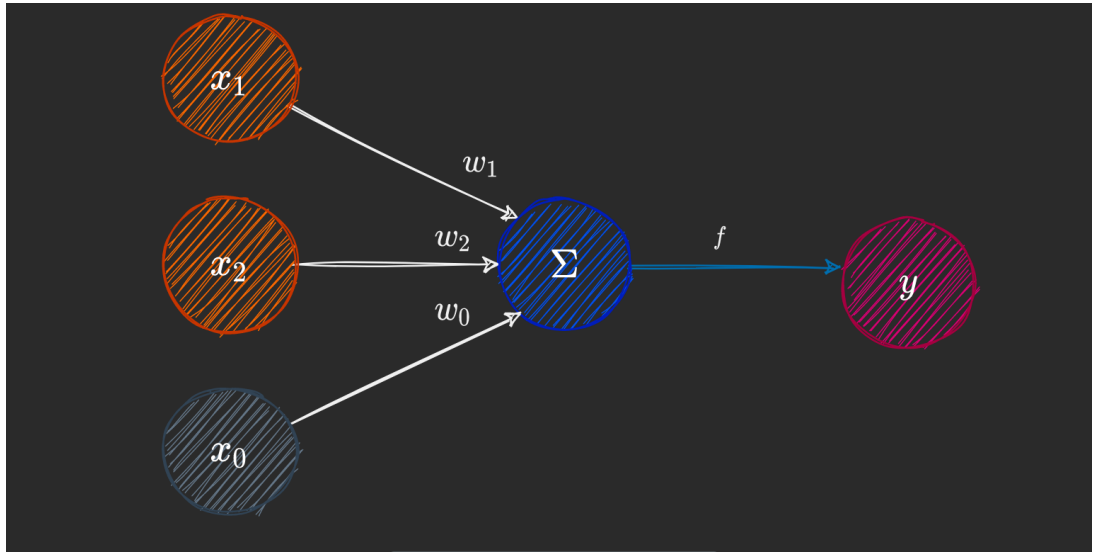


Artificial Neuron Model:
Perceptron input vs output





PERCEPTRON
Architectural components



Perceptron Components :

Input nodes $x_0 \dots x_n$

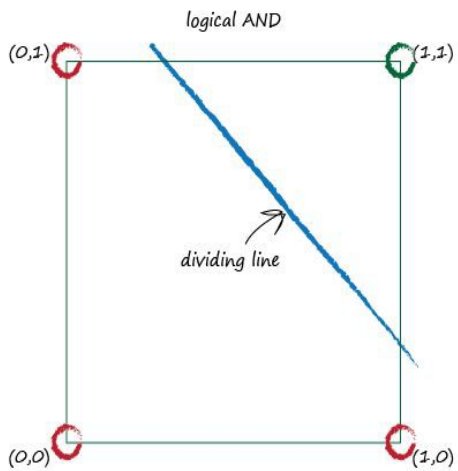
Output node y

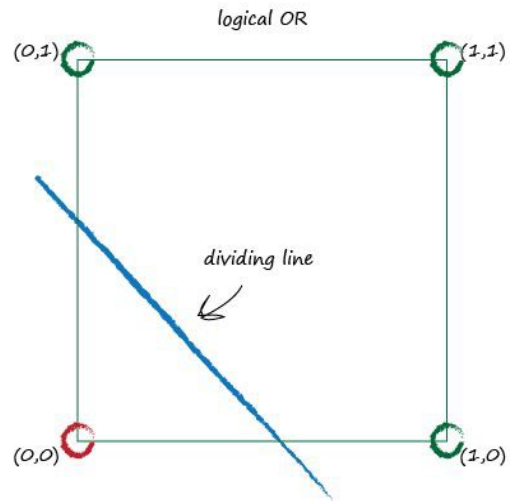
Weights $w_0 \dots w_n$

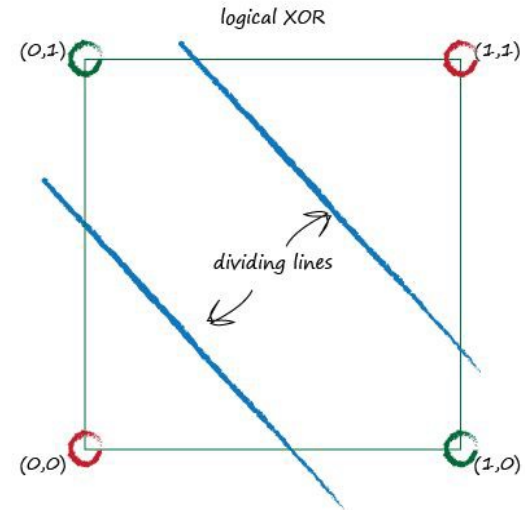
bias (=weight zero) w_0

Summation Σ (sigma)

Activation Function f


logical AND
(0,1)
(1,1)
dividing line
(0,0)
(1,0)


logical OR
(0,1)
(1,1)
dividing line
(0,0)
(1,0)


logical XOR
(0,1)
(1,1)
dividing lines
(0,0)
(1,0)

PERCEPTRON
Input /output relationship [logical AND]

Pseudo code:

Define number of training inputs. (n=2)

Create binary input value matrix
for **x** vector [x_1 x_2]

0 0
0 1
1 0
1 1

Define epoch (or batch) length (n=4)
Equals max number of binary input combinations of 2

Define teacher as a vector for the desired target values (n=4) to each input combination
[0 0 0 1]

Python code

```
############################
############################
############################
############################
############################
############################
training_inputs = []
training_inputs.append(np.array([1, 1]))
training_inputs.append(np.array([1, 0]))
training_inputs.append(np.array([0, 1]))
training_inputs.append(np.array([0, 0]))

teacher = np.array([1, 0, 0, 0])
```

PERCEPTRON
Input /output relationship [logical AND]

Pseudo code:

Define number of training inputs. (n=2)

Create binary input value matrix **(training dataset)**
for **x** vector [x_1 x_2]

0	0
0	1
1	0
1	1

Define epoch (or batch) length (n=4)
Equals max number of binary input combinations of 2

Define teacher as a vector for the desired target values (n=4) to each input combination
[0 0 0 1]

Epoch:

The number of epochs is a [hyper]parameter that defines the number times [iterations] that the perceptron will have to calculate output for the each of the input pairs of the training dataset

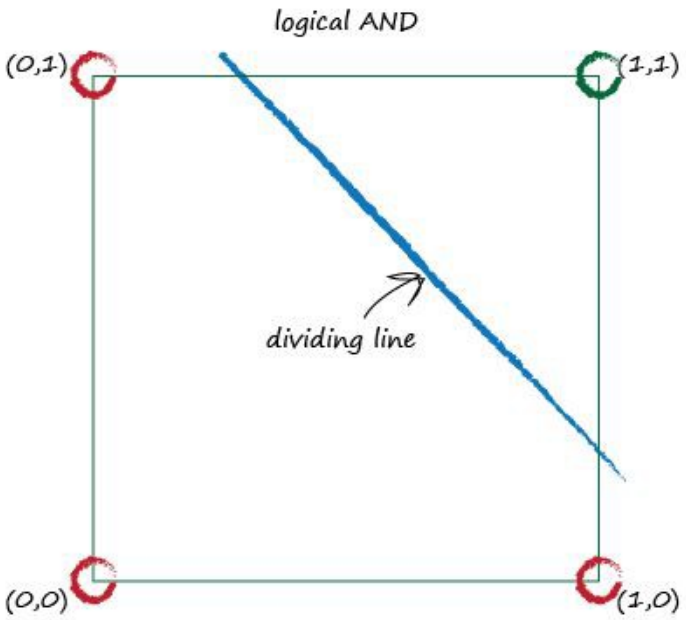
One Epoch
Defined by the ENTIRE input (training) dataset is passed forward only ONCE

=======> Logical AND perceptron
1 Epoch requires n=4 iterations

Perceptron
binary input vs output

Target [TEACHER]

Input x1	Input x2	AND y
0	0	0
0	1	0
1	0	0
1	1	1



Boolean Logic

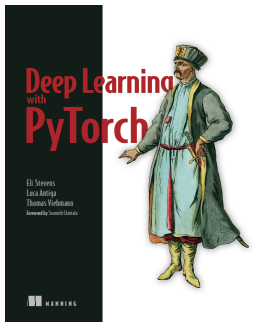
IF I have eaten my vegetables **AND** I am still hungry **THEN** I can have ice cream.

IF it's the weekend **OR** I am on annual leave **THEN** I'll go to the park.

Input A	Input B	AND	OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

MACHINE LEARNING [ML]

HOGESCHOOL ROTTERDAM



PERCEPTRON
Feed Forward Algorithm

Pseudo code:

Initialize the weights vector w to zeroes or random numbers

Compute DOT PRODUCT
of the binary input **x** vector $[x_1\ x_2]$
and binary weight **w** vector $[w_1\ w_2]$

Where bias b equals $w_{0\ \text{(weight zero)}}$

Apply the activation function F

PERCEPTRON
Class Python code

pseudo code:

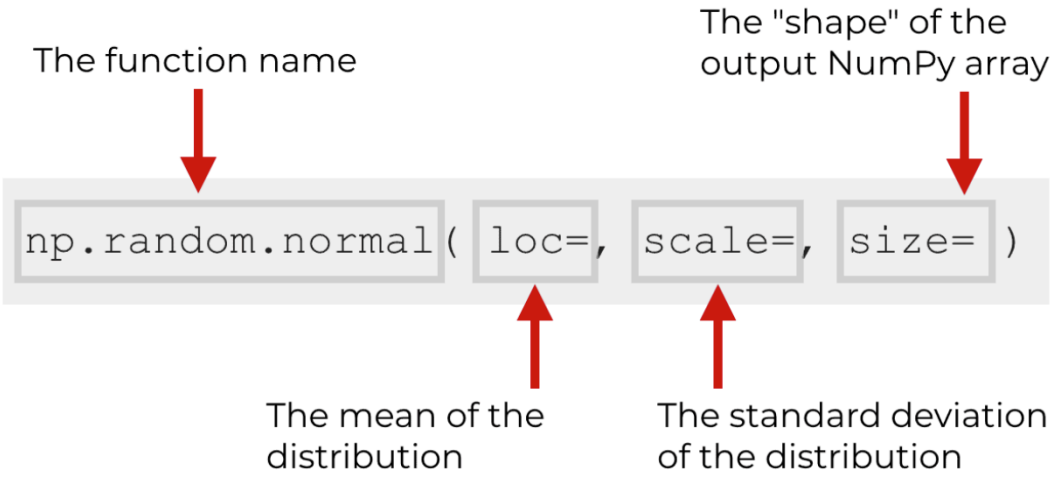
Define initializing object for the class named
Perceptron

Parameter settings
number of inputs x
number of epochs (or batches)
Learning rate

Initialization
Weights-vector (inclusive bias)
to zero's or random numbers

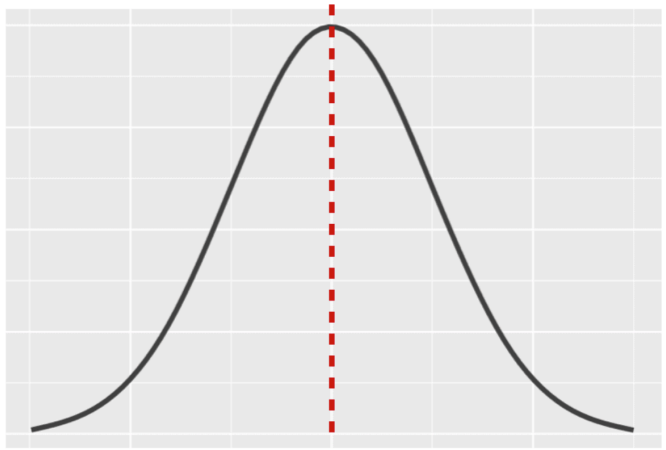
```
class Perceptron(object):

    def __init__(self, no_of_inputs, no_of_batches=10, learning_rate=0.01):
        self.no_of_batches = no_of_batches
        self.learning_rate = learning_rate
        self.weights = np.random.normal(0, 0.5, no_of_inputs + 1)
        self.output_bias = self.weights[0]
```

Randomization
Normal distribution
Python code

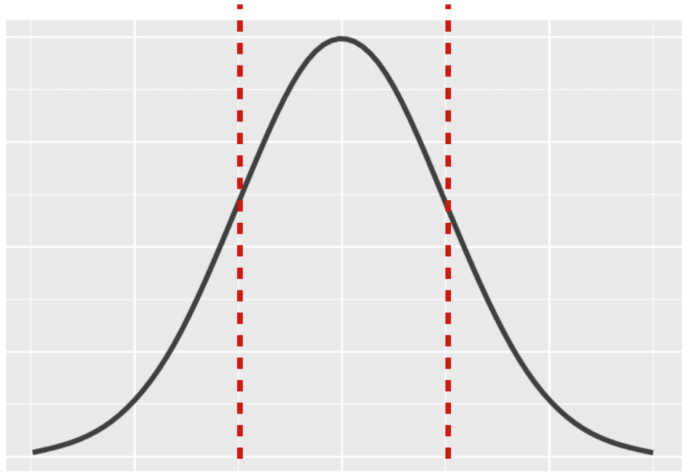



```
self.weights = np.random.normal(0, 0.5, no_of_inputs + 1)
```

The `loc` parameter sets the mean of the data



The `scale` parameter sets the standard deviation of the data



{01}
Fundamentals

The function name

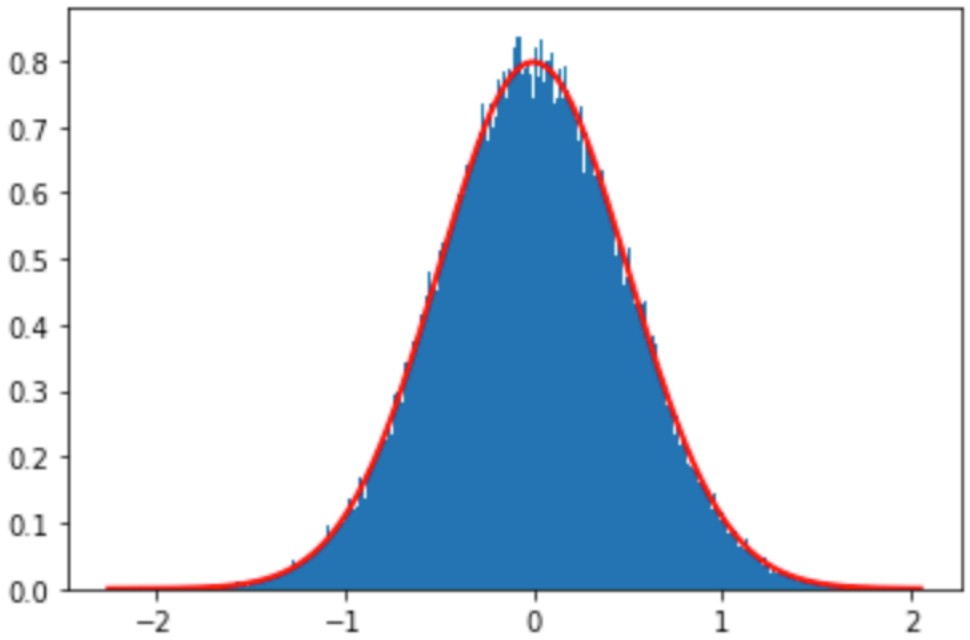
The "shape" of the output NumPy array

```
np.random.normal( loc=, scale=, size= )
```

The mean of the distribution

The standard deviation of the distribution

```python
import numpy as np # helps with the math
import matplotlib.pyplot as plt # to plot

mu, sigma = 0, 0.5 # mean and standard deviation
s = np.random.normal(0, 0.5, 100000)

count, bins, ignored = plt.hist(s, 300, density=True)
plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi))
* np.exp( - (bins - mu)**2
/ (2 * sigma**2) ),linewidth=2, color='r')
plt.show()
```



PERCEPTRON
Feed Forward maths

Feed Forward output calculation

pseudo code:

Compute DOT PRODUCT
of the binary input **x** vector
$[x_1\ x_2]$
and binary weight **w** vector
$[w_1\ w_2]$

Where bias b equals w_0 (weight zero)

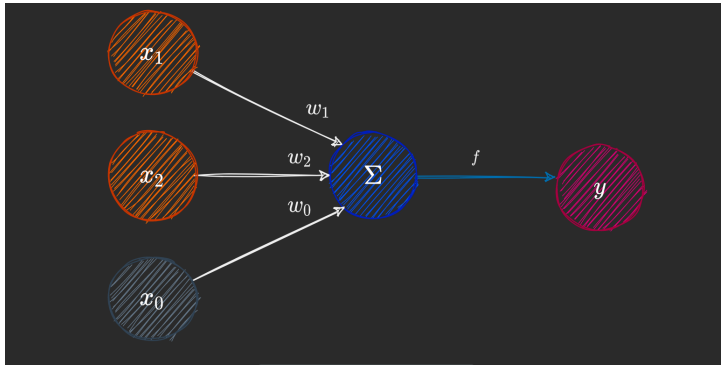
Apply the activation function f(**x**)

$$y = f(w \cdot X + b)$$

Feedforward computation:
(DOT PRODUCT is a form of summation

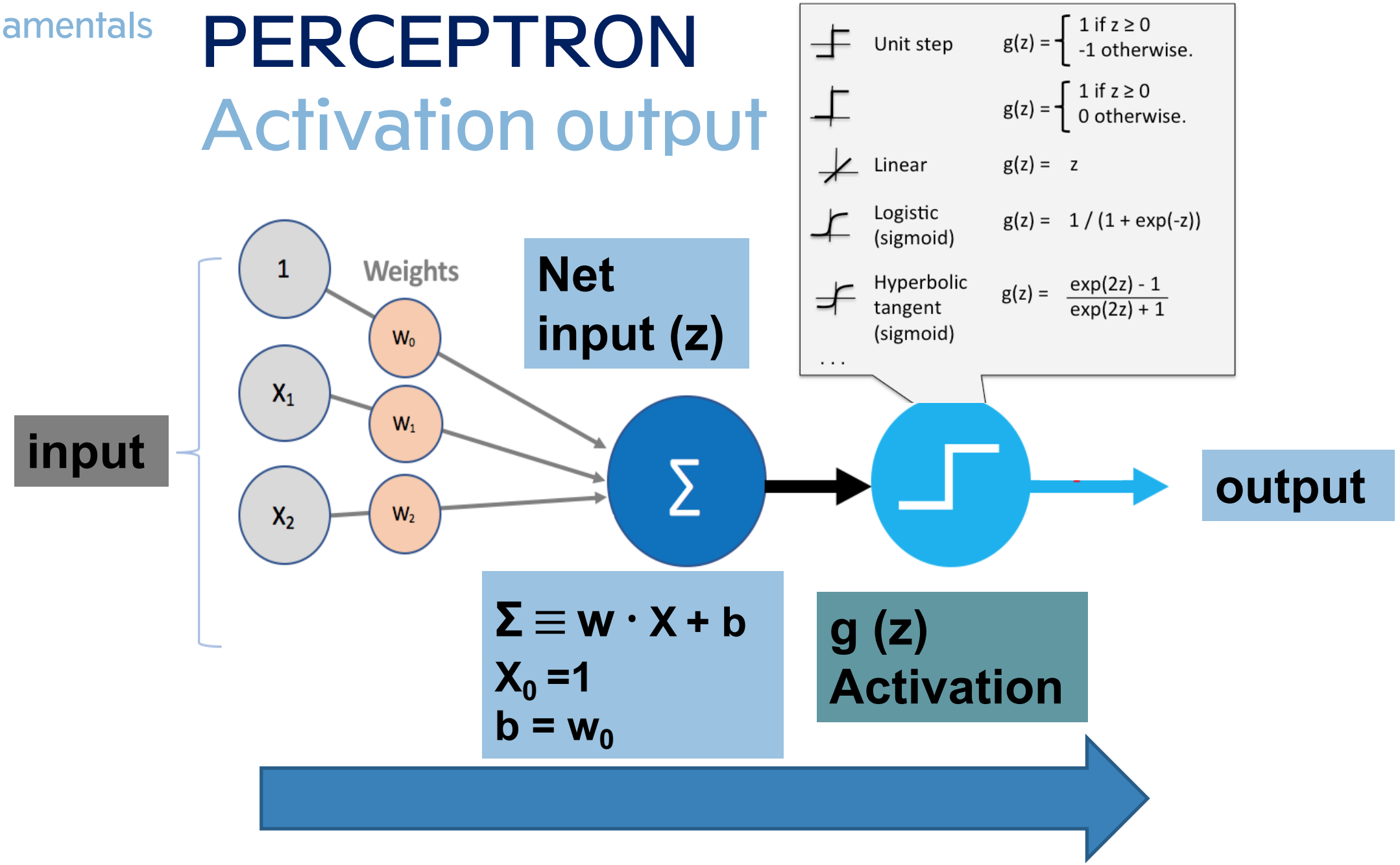
$$\Sigma \equiv \mathbf{w} \cdot \mathbf{X} + \mathbf{b}$$

$$\mathbf{X_0 = 1 \text{ and } b = w_0}$$

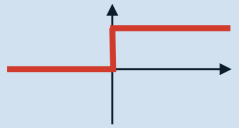
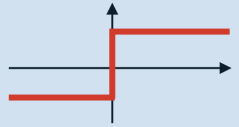
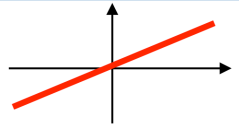
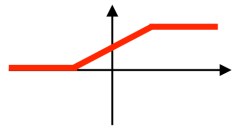
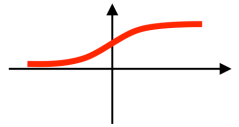
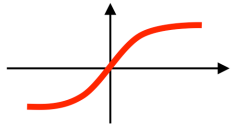


$$y = f(x_1 \cdot w_1 + x_2 \cdot w_2 + b)$$

PERCEPTRON
Activation output



Machine Learning
Activation functions math

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer NN	

PERCEPTRON
Class Python code

pseudo code:

**Define a predicted output through a STEP activation function
for the class named
Perceptron**

**During FEED FORWARD Calculation
Apply Activation function with dot multiplication from inputs and the weights
as net input**

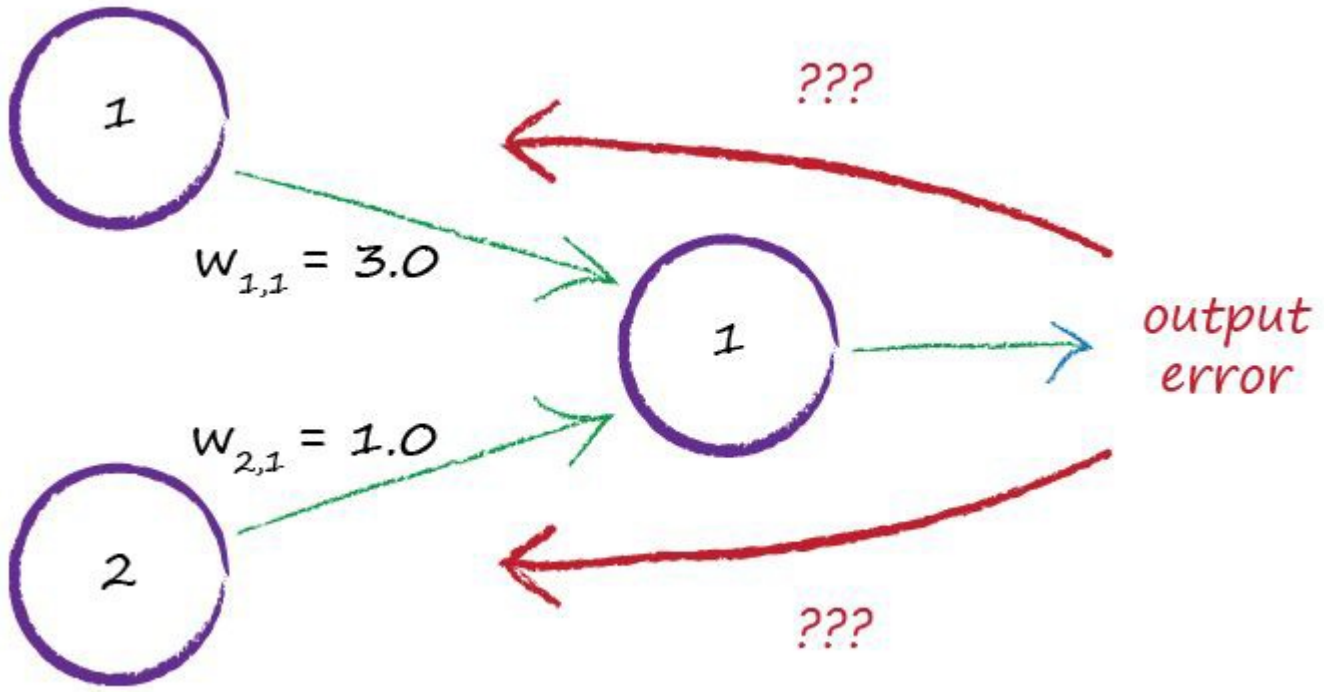
**If net input + bias is greater than 0 then activation is 1
otherwise make output 0**

```
def predict(self, inputs):
    summation = np.dot(inputs, self.weights[1:]) + self.output_bias
    if summation > 0:
      activation = 1
    else:
      activation = 0
    return activation
```

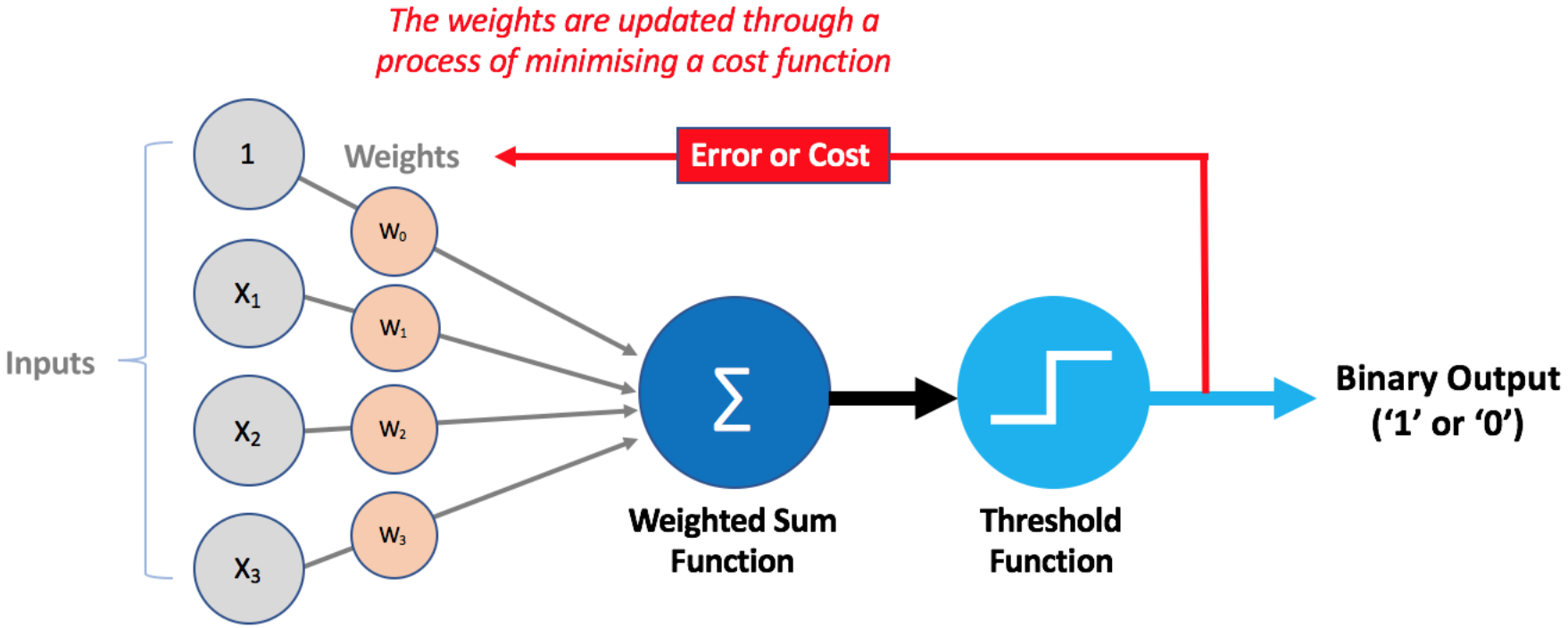
How do you get a Perceptron to learn?

needs iterative

FEED BACKWARD calculation == Training



Learning occurs through systematic adaptation of the weights
➔ FEED BACKWARD calculation🡐



PERCEPTRON

Learning algorithm maths calculates weight adaptation

Error (Loss or Cost) calculation

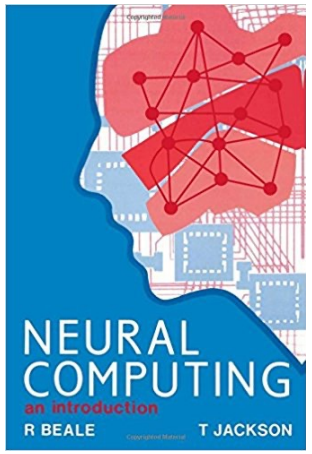
$$\Delta w = node \cdot (actual - computed)$$

weight adaptation

requires Δw and a learning rate parameter *lr*

$$w_{updated} = w_{old} + lr \cdot \Delta w$$

PERCEPTRON
Learning algorithm maths



3

The Basic Neuron

3.1 INTRODUCTION

In Chapter 1, we have examined the structure of the brain, and found it to be a highly developed mechanism that is relatively poorly understood, but capable of immensely impressive tasks. We have seen that many of the things that we would like computers to be able to do, the brain manages exceptionally well, and the idea behind neural computing is that by modelling the major features of the brain and its operation, we can produce computers that exhibit many of the useful properties of the brain.

We have noted the complexity of the structure of the brain; however, it can be viewed as a highly interconnected network of relatively simple processing elements. We need a model that can capture the important features of real neural systems in order that it will exhibit similar behaviour. However, the model must deliberately ignore many small effects, if it is to be simple enough to implement and understand. This extraction of a few features deemed important and disregard of all others is a general characteristic of modelling; the aim of a model is to produce a simplified version of a system which retains the same general behaviour, so that the system can be more easily understood.

3.2 MODELLING THE SINGLE NEURON

We will firstly consider the features of a single neuron and how we can model it. The basic function of a biological neuron is to add

39

Perceptron Learning Algorithm

1. Initialise weights and threshold
Define $w_i(t), (0 \leq i \leq n)$, to be the weight from input i at time t, and θ to be the threshold value in the output node. Set w_0 to be $-\theta$, the bias, and x_0 to be always 1.

Set $w_i(0)$ to small random values, thus initialising all the weights and the threshold.

2. Present input and desired output
Present input $x_0, x_1, x_2, \ldots, x_n$ and desired output $d(t)$

3. Calculate actual output

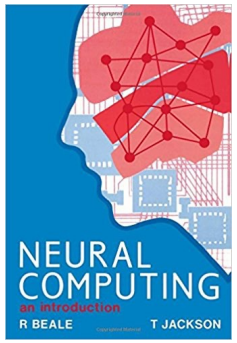
$$y(t) = f_h\left[\sum_{i=0}^{n} w_i(t)x_i(t)\right]$$

4. Adapt weights

$$\begin{aligned} \text{if correct} \quad & w_i(t+1) = w_i(t) \\ \text{if output 0, should be 1 (class A)} \quad & w_i(t+1) = w_i(t) + x_i(t) \\ \text{if output 1, should be 0 (class B)} \quad & w_i(t+1) = w_i(t) - x_i(t) \end{aligned}$$

Note that weights are unchanged if the net makes the correct decision. Also, weights are not adjusted on input lines which do not contribute to the incorrect response, since each weight is adjusted by the value of the input on that line, x_i, which would be zero.

PERCEPTRON
Learning algorithm maths



4. Adapt weights—modified version

$$
\begin{array}{rrcl}
\text{if correct} & w_i(t+1) & = & w_i(t) \\
\text{if output 0, should be 1 (class A)} & w_i(t+1) & = & w_i(t) + \eta x_i(t) \\
\text{if output 1, should be 0 (class B)} & w_i(t+1) & = & w_i(t) - \eta x_i(t)
\end{array}
$$

where $0 \leq \eta \leq 1$, a positive gain term that controls the adaption rate.

Widrow-Hoff
Δ Delta-rule

4. Adapt weights—Widrow-Hoff delta rule

$$
\begin{aligned}
\Delta &= d(t) - y(t) \\
w_i(t+1) &= w_i(t) + \eta \Delta x_i(t) \\
d(t) &= \begin{cases} +1, & \text{if input from class A} \\ 0, & \text{if input from class B} \end{cases}
\end{aligned}
$$

where $0 \leq \eta \leq 1$, a positive gain function that controls the adaption rate

Neural Networks using the Widrow-Hoff delta rule are called (M)ADALINE: (many) Adaptive Linear Neurons

pseudo code:

**Define training function
for the class named Perceptron**

**Define iteration cycle and keep
track of the number of iteration**

**Do this for all possible input/ouput
combinations
(4 iterations =
1 epoch or training batch)**

**Calculate error
by subtracting
(predicted) activated feedforward
output from the desired target label
value (teacher) multiplied by a
given learning-rate and input value**

**Weights are adapted each iteration
by summing the old weight with the
calculated error value**

PERCEPTRON
Class Python code

```
def train(self, training_inputs, labels):
    interation_number = 0
    bach_number = 0
    for _ in range(self.no_of_batches):

        for inputs, label in zip(training_inputs, labels):
            prediction = self.predict(inputs)
            self.weights[1:] += self.learning_rate * (label - prediction) * inputs
            self.output_bias += self.learning_rate * (label - prediction)
            print(label, prediction)
```

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

class Perceptron(object):

    def __init__(self, no_of_inputs, no_of_training_epochs=20, learning_rate=0.01):
        self.no_of_training_epochs = no_of_training_epochs
        self.learning_rate = learning_rate
        self.weights = np.random.normal(0, 10, no_of_inputs + 1)
        self.bias = self.weights[0]
        self.epoch_list=[]
        self.error_history=[]


    def step_function(self, inputs):
        netto_summation = np.dot(inputs, self.weights[1:]) + self.bias
        if netto_summation > 0:
          activation = 1
        else:
          activation = 0
        return activation

    def train(self, training_inputs, teacher_labels):
        epoch =0;
        i =0;
        for _ in range(self.no_of_training_epochs):
          epoch+=1
          for inputs, teacher in zip(training_inputs, teacher_labels):
              activated_perceptron_output = self.step_function(inputs)
              error = (teacher - activated_perceptron_output)
              self.weights[1:] += self.learning_rate * (error) * inputs
              self.bias += self.learning_rate  * error
              i+=1
              self.epoch_list.append(epoch)
              self.error_history.append(np.sum(np.abs(error)))
```

PYTHON PERCEPTRON MODEL code

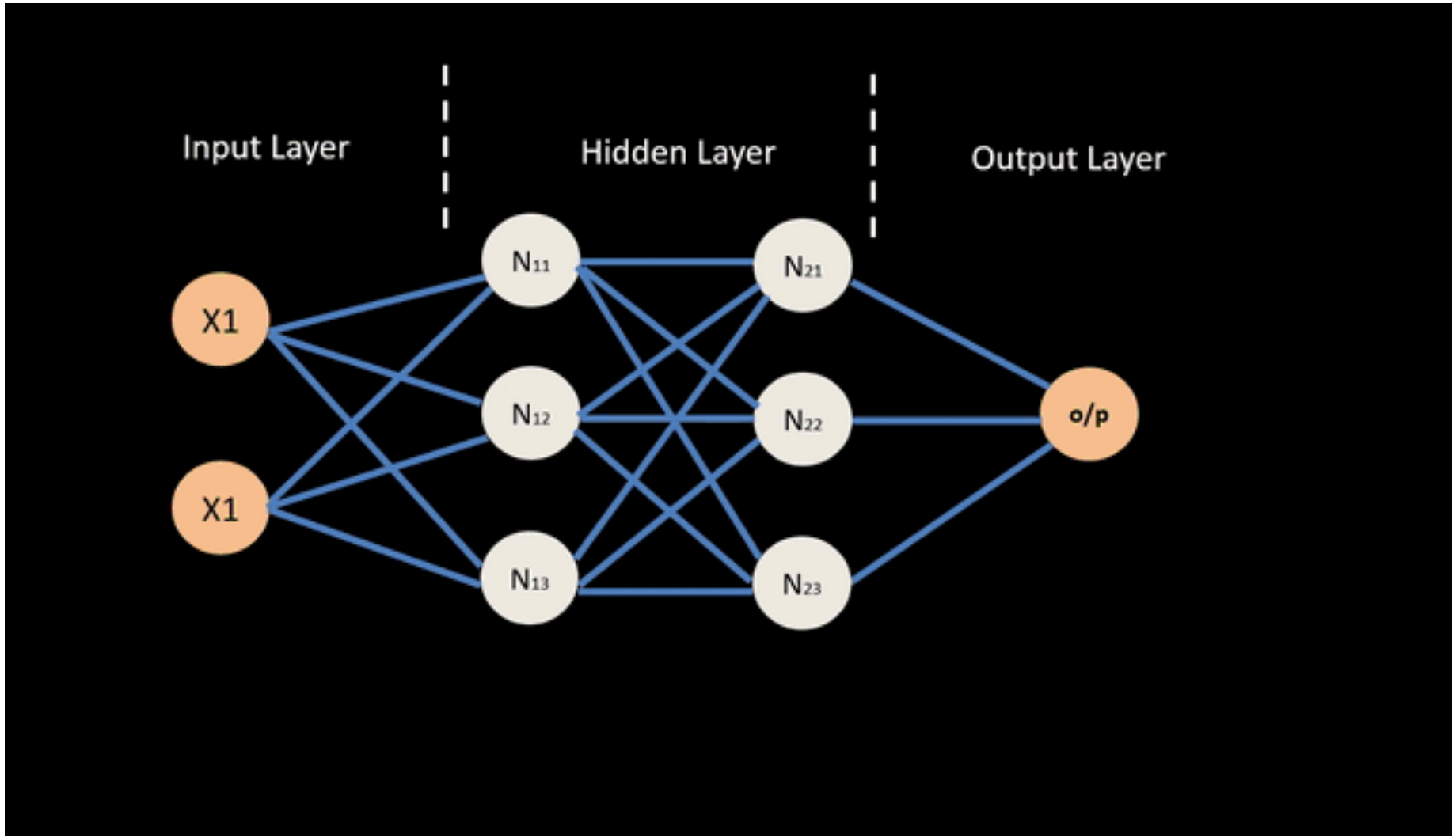
import Numpy as np
import Matlibplot.pyplot as plt

Where are input & ouput specified?

Where are the weights initialized?

How is the Forward computation specified?

To Do:
Change the code as to print
Interation number
Bach number
Error
And weights


Input Layer
Hidden Layer
Output Layer
X1
X1
N11
N12
N13
N21
N22
N23
o/p

PYTHON
INPUT / OUTPUT relationship code

```
import Numpy as np
import Matlibplot.pyplot as plt

import numpy as np

inputs = []
inputs.append(np.array([1, 1]))
inputs.append(np.array([1, 0]))
inputs.append(np.array([0, 1]))
inputs.append(np.array([0, 0]))

teacher = np.array([1, 0, 0, 0])
```

USING ZIP

```
print([inputs for teacher in zip(inputs, teacher)])

print([i for i in zip(inputs, teacher)])
```

{01}
ANSWERS

ZIP function
Coded Python Perceptron class object

```python
import numpy as np

inputs = []
inputs.append(np.array([1, 1]))
inputs.append(np.array([1, 0]))
inputs.append(np.array([0, 1]))
inputs.append(np.array([0, 0]))

teacher = np.array([1, 0, 0, 0])
print([inputs for teacher in zip(inputs, teacher)])
print([i for i in zip(inputs, teacher)])

```

```
[[array([1, 1]), array([1, 0]), array([0, 1]), array([0, 0])], [array([1, 1]), array([1, 0]), array([0, 1]),
[(array([1, 1]), 1), (array([1, 0]), 0), (array([0, 1]), 0), (array([0, 0]), 0)]
```

```
[[array([1, 1]), array([1, 0]), array([0, 1]), array([0, 0])],
[array([1, 1]), array([1, 0]), array([0, 1]), array([0, 0])],
[array([1, 1]), array([1, 0]), array([0, 1]), array([0, 0])],
[array([1, 1]), array([1, 0]), array([0, 1]), array([0, 0])]]
```

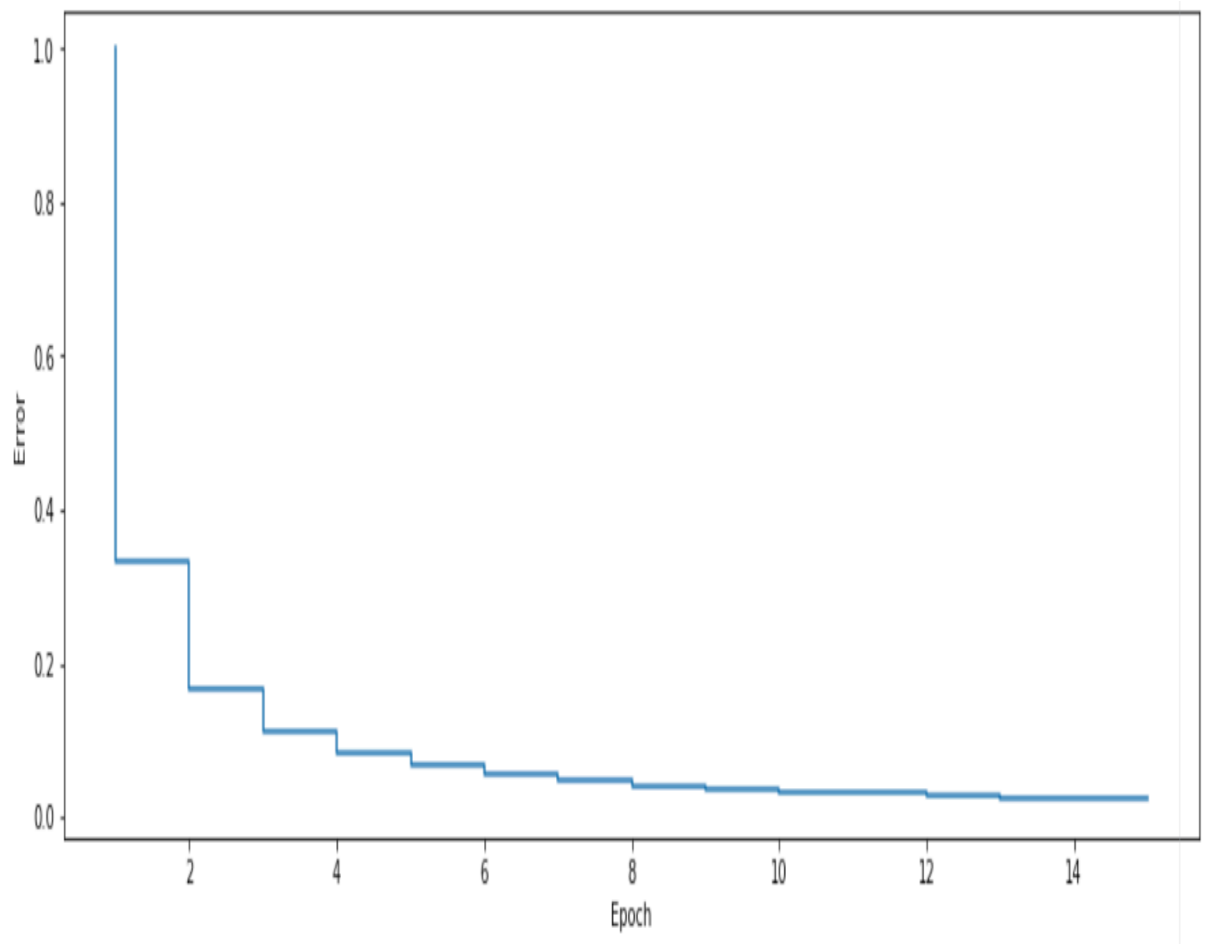
Training of the
Coded Python Perceptron class object

```
training_inputs = []
training_inputs.append(np.array([1, 1]))
training_inputs.append(np.array([1, 0]))
training_inputs.append(np.array([0, 1]))
training_inputs.append(np.array([0, 0]))

teacher = np.array([1, 0, 0, 0])


NN=perceptron = Perceptron(2,15,0.1)
NN.train(training_inputs, teacher)
```

{01}
ANSWERS

Evidence of learning through training of the Coded Python Perceptron class object

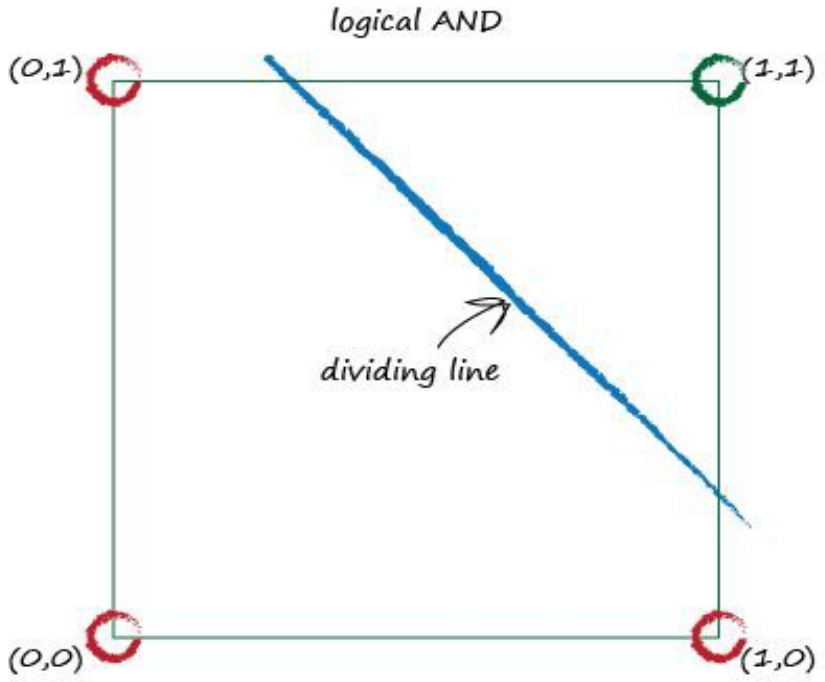
```
    def train(self, training_inputs, teacher_labels):
        epoch =0;
        i =0;
        for _ in range(self.no_of_training_epochs):
          epoch+=1
          for inputs, teacher in zip(training_inputs, teacher_labels):
              activated_perceptron_output = self.step_function(inputs)
              error = (teacher - activated_perceptron_output)
              self.weights[1:] += self.learning_rate * (error) * inputs
              self.bias += self.learning_rate  * error
              #print(teacher, activated_perceptron_output)
              i+=1
              self.epoch_list.append(epoch)
              self.error_history.append(np.sum(np.abs(error)))

#####################################
#####################################
#####################################
#####################################
#####################################
training_inputs = []
training_inputs.append(np.array([1, 1]))
training_inputs.append(np.array([1, 0]))
training_inputs.append(np.array([0, 1]))
training_inputs.append(np.array([0, 0]))

teacher = np.array([1, 0, 0, 0])


NN=perceptron = Perceptron(2,15,0.1)
NN.train(training_inputs, teacher)


plt.figure(figsize=(15,5))
plt.plot(NN.epoch_list, 1/np.cumsum(NN.error_history))
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.show()
```



How do you determine if a Perceptron has learned the input/output relationship?

Target label [TEACHER]

Input x1	Input x2	AND y
0	0	0
0	1	0
1	0	0
1	1	1

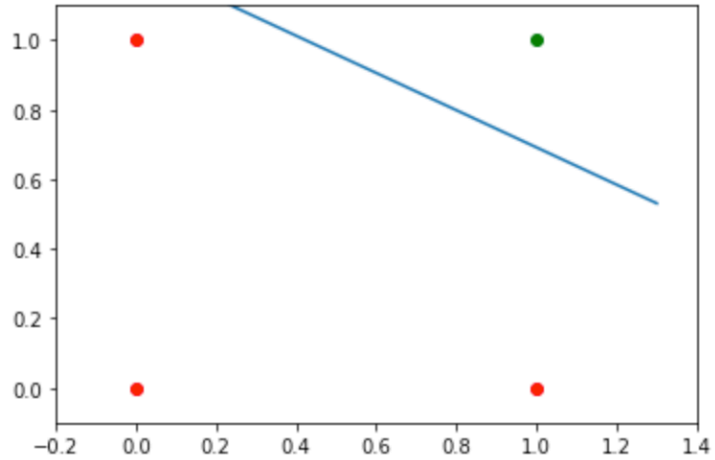


{01}
ANSWERS

Evidence of learning through training of the Coded Python Perceptron class object

Input (x) | Output (y) Space

```
training_inputs = []
training_inputs.append(np.array([1, 1]))
training_inputs.append(np.array([1, 0]))
training_inputs.append(np.array([0, 1]))
training_inputs.append(np.array([0, 0]))

teacher = np.array([1, 0, 0, 0])


NN=perceptron = Perceptron(2,15,0.1)
NN.train(training_inputs, teacher)
m = -(NN.weights[1] / NN.weights[2])
c = -(NN.output_bias / NN.weights[2])

plt.figure(figsize=(15,5))
fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.4
X = np.arange(xmin, xmax, 0.1)
ax.scatter(0, 0, color="r")
ax.scatter(0, 1, color="r")
ax.scatter(1, 0, color="r")
ax.scatter(1, 1, color="g")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])


print(m, c)
ax.plot(X, m * X + c )
plt.plot()
```



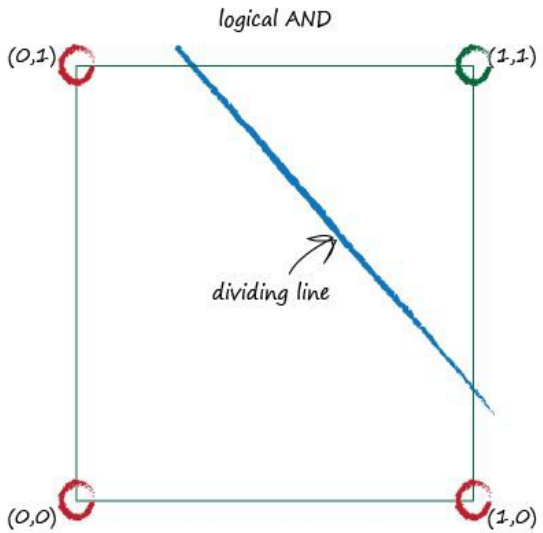
$$y = \frac{w_1}{w_2} * x + \frac{w_0}{w_2}$$

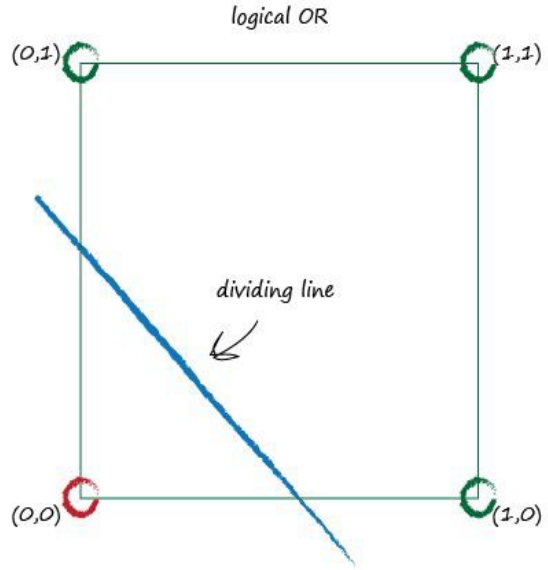
$$Y = m*x+c$$

$$Class_1 : w \cdot X + b \geq 0$$

$$Class_2 : w \cdot X + b < 0$$

Input (x) | Output (y) Space





$$Class_1 : w \cdot X + b \geq 0$$

$$Class_2 : w \cdot X + b < 0$$

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

class Perceptron(object):

    def __init__(self, no_of_inputs, no_of_training_epochs=20, learning_rate=0.01):
        self.no_of_training_epochs = no_of_training_epochs
        self.learning_rate = learning_rate
        self.weights = np.random.normal(0, 10, no_of_inputs + 1)
        self.bias = self.weights[0]
        self.epoch_list=[]
        self.error_history=[]


    def step_function(self, inputs):
        netto_summation = np.dot(inputs, self.weights[1:]) + self.bias
        if netto_summation > 0:
          activation = 1
        else:
          activation = 0
        return activation

    def train(self, training_inputs, teacher_labels):
        epoch =0;
        i =0;
        for _ in range(self.no_of_training_epochs):
          epoch+=1
          for inputs, teacher in zip(training_inputs, teacher_labels):
              activated_perceptron_output = self.step_function(inputs)
              error = (teacher - activated_perceptron_output)
              self.weights[1:] += self.learning_rate * (error) * inputs
              self.bias += self.learning_rate  * error
              i+=1
              self.epoch_list.append(epoch)
              self.error_history.append(np.sum(np.abs(error)))
```

PYTHON PERCEPTRON MODEL code

import Numpy as np
import Matlibplot.pyplot as plt

Where are input & ouput specified?

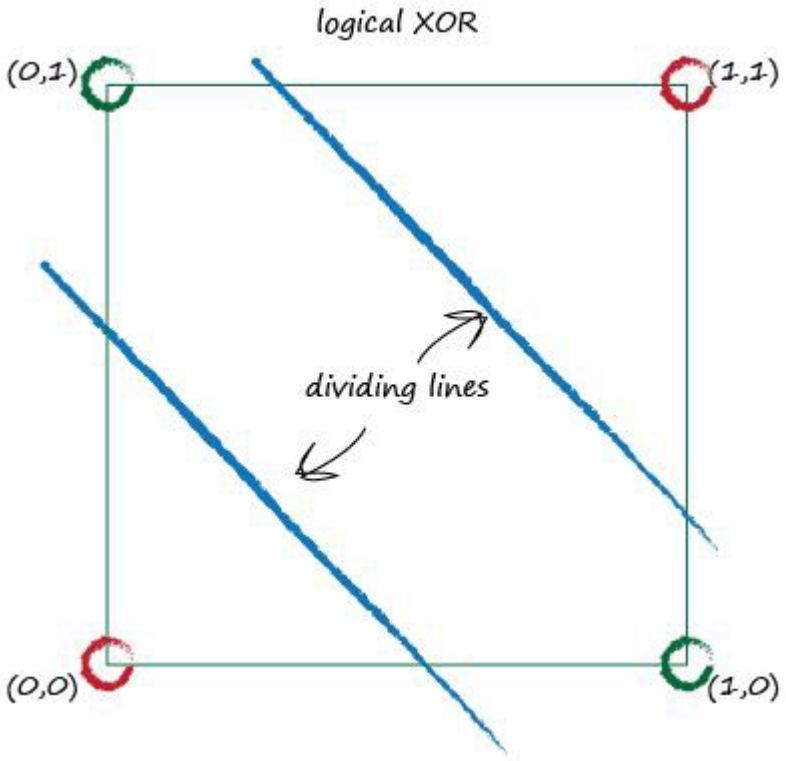
Where are the weights initialized?

How is the Forward computation specified?

To Do:
Change the code as to print
Interation number
Bach number
Error
And weights

Code Python Perceptron class object for XOR PROBLEM

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

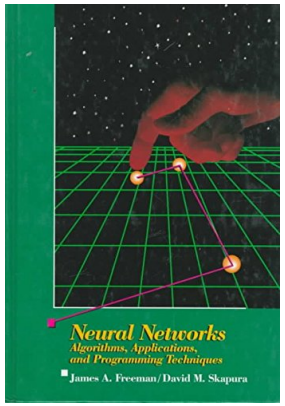


{04}
Do it Yourself

HOGESCHOOL ROTTERDAM

Write code in Pyton
that solves the XOR problem based
on the perceptron neural network architecture using the
BACKPROPAGATION learning rule algorithm.

Use the description of Backpropagation as specified in:

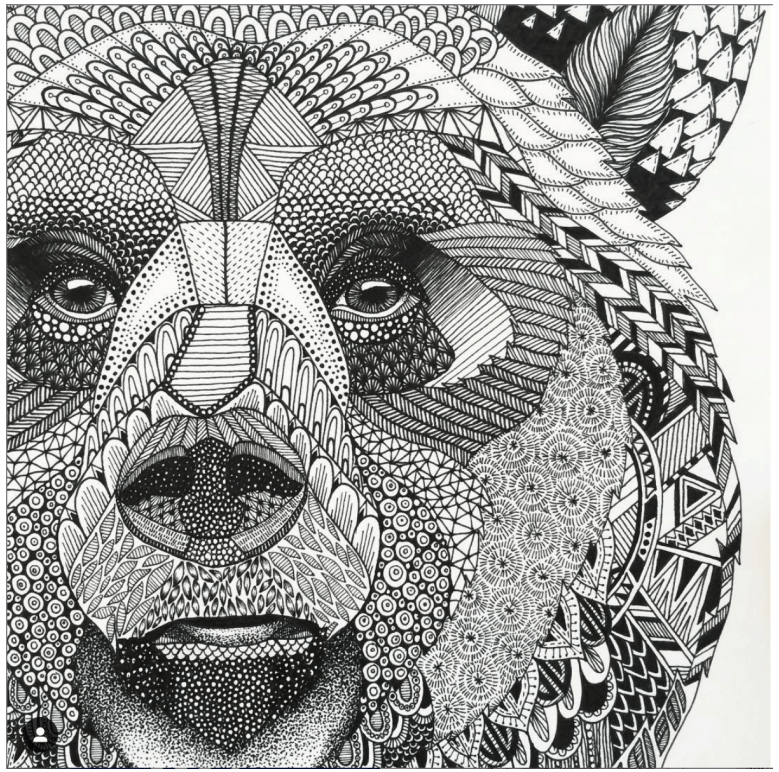


{04}

Do it Yourself

HOGESCHOOL ROTTERDAM

Use
import Numpy as np
import Matlibplot.pyplot as plt



Where are input & ouput specified?

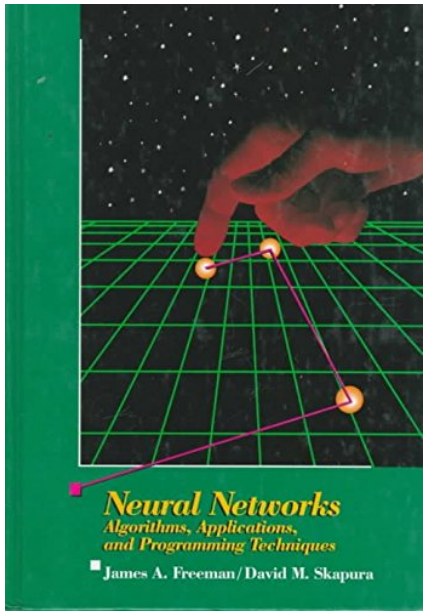
Where are the weights + biases initialized?

What type of Activation function must be used.

How is the Forward computation specified?

How is Backward adaptation specified (BACKPROPAGATION)

Demonstate that your network learns and has solved the XOR input-output relationship.



CHAPTER

3

Backpropagation

There are many potential computer applications that are difficult to implement because there are many problems unsuited to solution by a sequential process. Applications that must perform some complex data translation, yet have no predefined mapping function to describe the translation process, or those that must provide a "best guess" as output when presented with noisy input data are but two examples of problems of this type.

An ANS that we have found to be useful in addressing problems requiring recognition of complex patterns and performing nontrivial mapping functions is the backpropagation network (BPN), formalized first by Werbos [11], and later by Parker [8] and by Rummelhart and McClelland [7]. This network, illustrated generically in Figure 3.1, is designed to operate as a multilayer, feedforward network, using the supervised mode of learning.

The chapter begins with a discussion of an example of a problem mapping character image to ASCII, which appears simple, but can quickly overwhelm traditional approaches. Then, we look at how the backpropagation network operates to solve such a problem. Following that discussion is a detailed derivation of the equations that govern the learning process in the backpropagation network. From there, we describe some practical applications of the BPN as described in the literature. The chapter concludes with details of the BPN software simulator within the context of the general design given in Chapter 1.

3.1 THE BACKPROPAGATION NETWORK

To illustrate some problems that often arise when we are attempting to automate complex pattern-recognition applications, let us consider the design of a computer program that must translate a 5 x 7 matrix of binary numbers representing the bit-mapped pixel image of an alphanumeric character to its equivalent eight-bit ASCII code. This basic problem, pictured in Figure 3.2, appears to be relatively trivial at first glance. Since there is no obvious mathematical function

1. Apply the input vector, $\mathbf{x}_p = (x_{p1}, x_{p2}, \ldots, x_{pN})^t$ to the input units.
2. Calculate the net-input values to the hidden layer units:

$$\text{net}_{pj}^h = \sum_{i=1}^{N} w_{ji}^h x_{pi} + \theta_j^h$$

3. Calculate the outputs from the hidden layer:

$$i_{pj} = f_j^h(\text{net}_{pj}^h)$$

4. Move to the output layer. Calculate the net-input values to each unit:

$$\text{net}_{pk}^o = \sum_{j=1}^{L} w_{kj}^o i_{pj} + \theta_k^o$$

5. Calculate the outputs:

$$o_{pk} = f_k^o(\text{net}_{pk}^o)$$

6. Calculate the error terms for the output units:

$$\delta_{pk}^o = (y_{pk} - o_{pk}) f_k^{o\prime}(\text{net}_{pk}^o)$$

7. Calculate the error terms for the hidden units:

$$\delta_{pj}^h = f_j^{h\prime}(\text{net}_{pj}^h) \sum_k \delta_{pk}^o w_{kj}^o$$

   Notice that the error terms on the hidden units are calculated *before* the connection weights to the output-layer units have been updated.

8. Update weights on the output layer:

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \eta \delta_{pk}^o i_{pj}$$

9. Update weights on the hidden layer:

$$w_{ji}^h(t+1) = w_{ji}^h(t) + \eta \delta_{pj}^h x_i$$

   The order of the weight updates on an individual layer is not important.

   Be sure to calculate the error term

$$E_p = \frac{1}{2} \sum_{k=1}^{M} \delta_{pk}^2$$

since this quantity is the measure of how well the network is learning. When the error is acceptably small for each of the training-vector pairs, training can be discontinued.

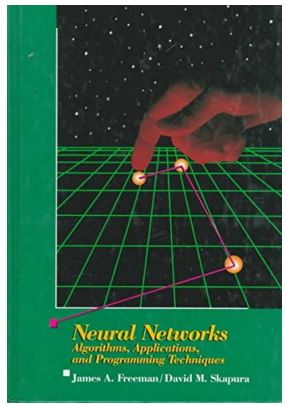


Figure 3.3 serves as the reference for most of the discussion. The BPN is a layered, feedforward network that is fully interconnected by layers. Thus, there are no feedback connections and no connections that bypass one layer to go directly to a later layer. Although only three layers are used in the discussion, more than one hidden layer is permissible.

A neural network is called a **mapping network** if it is able to compute some functional relationship between its input and its output. For example, if the input to a network is the value of an angle, and the output is the cosine of that angle, the network performs the mapping $9 \rightarrow \cos(\theta)$. For such a simple function, we do not need a neural network; however, we might want to perform a complicated mapping where we do not know how to describe the functional relationship in advance, but we do know of examples of the correct mapping.

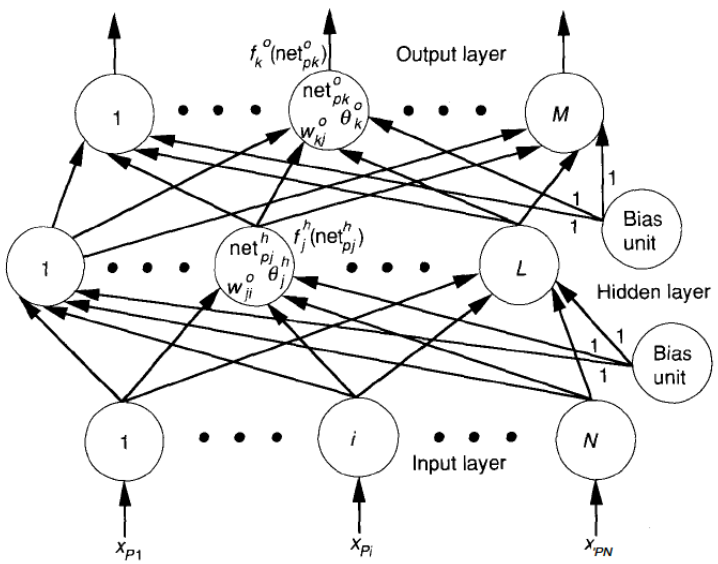


Figure 3.3 The three-layer BPN architecture follows closely the general network description given in Chapter 1. The bias weights, θ_j^h, and θ_k^o, and the bias units are optional. The bias units provide a fictitious input value of 1 on a connection to the bias weight. We can then treat the bias weight (or simply, bias) like any other weight: It contributes to the net-input value to the unit, and it participates in the learning process like any other weight.

In this situation, the power of a neural network to discover its own algorithms is extremely useful.

Suppose we have a set of P vector-pairs, $(\mathbf{x}_1, \mathbf{y}_1)$, $(\mathbf{x}_2, \mathbf{y}_2)$,..., (xp,yp), which are examples of a functional mapping $\mathbf{y} = \phi(\mathbf{x}) : \mathbf{x} \in \mathbf{R}^N, \mathbf{y} \in \mathbf{R}^M$. We want to train the network so that it will learn an approximation $\mathbf{o} = \mathbf{y}' = \phi'(\mathbf{x})$. We shall derive a method of doing this training that usually works, provided the training-vector pairs have been chosen properly and there is a sufficient number of them. (Definitions of *properly* and *sufficient* will be given in Section 33.) Remember that learning in a neural network means finding an appropriate set of weights. The learning technique that we describe here resembles the problem of finding the equation of a line that best fits a number of known points. Moreover, it is a generalization of the LMS rule that we discussed in Chapter 2. For a line-fitting problem, we would probably use a least-squares approximation. Because the relationship we are trying to map is likely to be nonlinear, as well as multidimensional, we employ an iterative version of the simple least-squares method, called a *steepest-descent technique*.

To begin, let's review the equations for information processing in the three-layer network in Figure 3.3. An input vector, $\mathbf{x}_p = (x_{p1}, x_{p2}, \ldots, x_{pN})^t$ is applied to the input layer of the network. The input units distribute the values to the hidden-layer units. The net input to the jth hidden unit is

$$\text{net}_{pj}^h = \sum_{i=1}^{N} w_{ji}^h x_{pi} + \theta_j^h \tag{3.1}$$

where w_{ji}^h is the weight on the connection from the ith input unit, and θ_j^h is the bias term discussed in Chapter 2. The "h" superscript refers to quantities on the hidden layer. Assume that the activation of this node is equal to the net input; then, the output of this node is

$$i_{pj} = f_j^h(\text{net}_{pj}^h) \tag{3.2}$$

The equations for the output nodes are

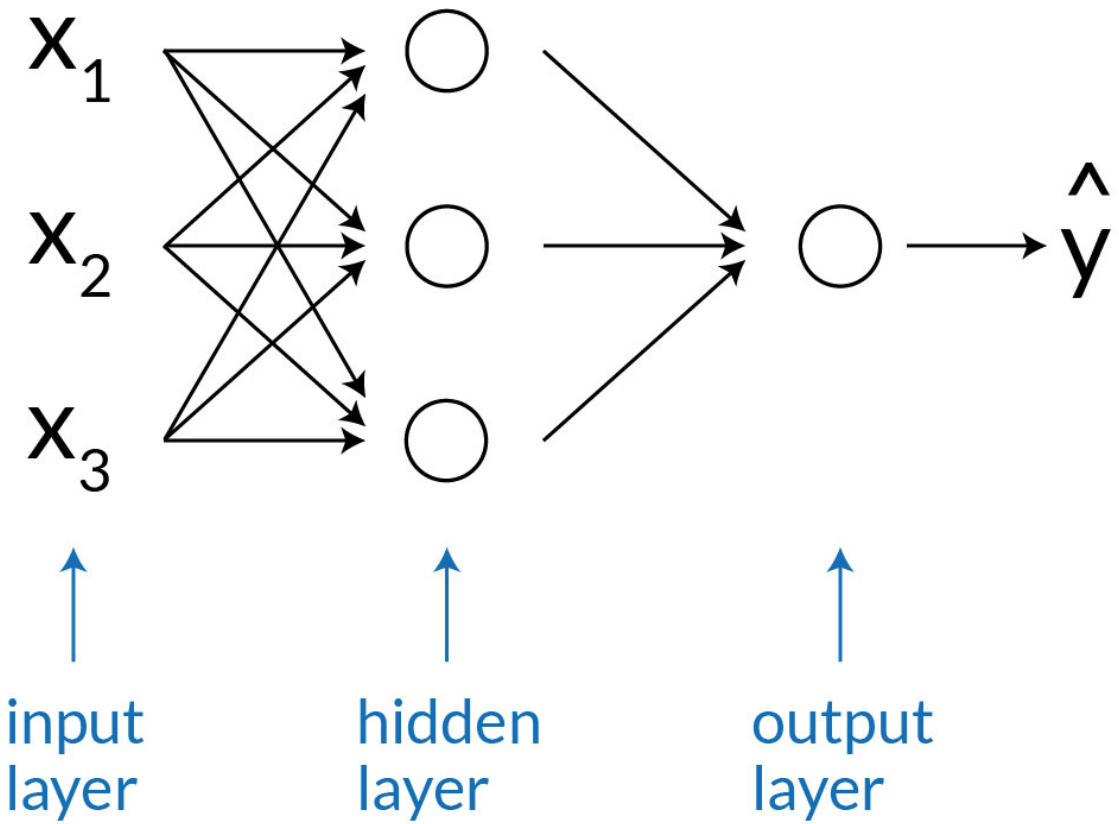
$$\text{net}_{pk}^o = \sum_{j=1}^{L} w_{kj}^o i_{pj} + \theta_k^o \tag{3.3}$$

$$o_{pk} = f_k^o(\text{net}_{pk}^o) \tag{3.4}$$

where the "o" superscript refers to quantities on the output layer.

The initial set of weight values represents a first guess as to the proper weights for the problem. Unlike some methods, the technique we employ here does not depend on making a *good* first guess. There are guidelines for selecting the initial weights, however, and we shall discuss them in Section 3.3. The basic procedure for training the network is embodied in the following description:

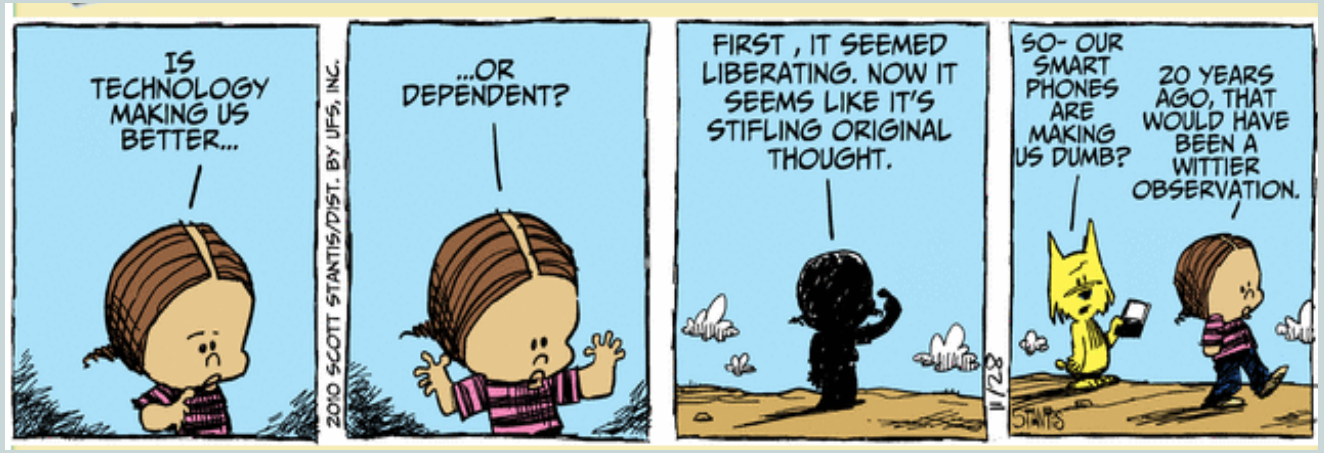
Code Python Perceptron class object
for XOR PROBLEM requires hidden layer



Example Python Code
Perceptron class object for XOR PROBLEM

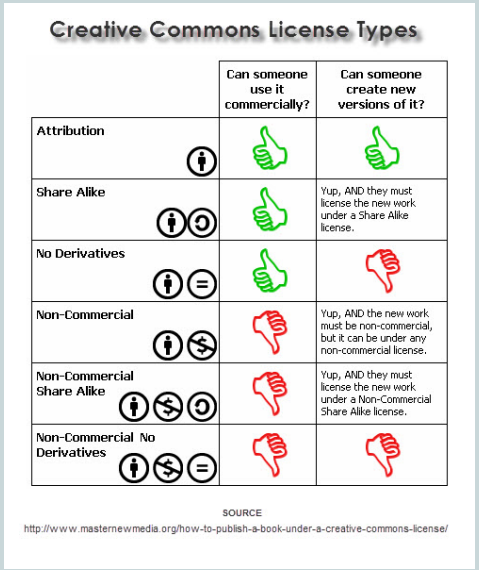
```python
import numpy as np
#np.random.seed(0)

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

def sigmoid_derivative(x):
    return x * (1 - x)

#Input datasets
inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
expected_output = np.array([[0],[1],[1],[0]])

epochs = 10000
lr = 0.1
inputLayerNeurons, hiddenLayerNeurons, outputLayerNeurons = 2,2,1

#Random weights and bias initialization
hidden_weights = np.random.uniform(size=(inputLayerNeurons,hiddenLayerNeurons))
hidden_bias =np.random.uniform(size=(1,hiddenLayerNeurons))
output_weights = np.random.uniform(size=(hiddenLayerNeurons,outputLayerNeurons))
output_bias = np.random.uniform(size=(1,outputLayerNeurons))

print("Initial hidden weights: ",end='')
print(*hidden_weights)
print("Initial hidden biases: ",end='')
print(*hidden_bias)
print("Initial output weights: ",end='')
print(*output_weights)
print("Initial output biases: ",end='')
print(*output_bias)
```

```python
#Training algorithm
for _ in range(epochs):
    #Forward Propagation
    hidden_layer_activation = np.dot(inputs,hidden_weights)
    hidden_layer_activation += hidden_bias
    hidden_layer_output = sigmoid(hidden_layer_activation)

    output_layer_activation = np.dot(hidden_layer_output,output_weights)
    output_layer_activation += output_bias
    predicted_output = sigmoid(output_layer_activation)

    #Backpropagation
    error = expected_output - predicted_output
    d_predicted_output = error * sigmoid_derivative(predicted_output)

    error_hidden_layer = d_predicted_output.dot(output_weights.T)
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

    #Updating Weights and Biases
    output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
    output_bias += np.sum(d_predicted_output,axis=0,keepdims=True) * lr
    hidden_weights += inputs.T.dot(d_hidden_layer) * lr
    hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) * lr

print("Final hidden weights: ",end='')
print(*hidden_weights)
print("Final hidden bias: ",end='')
print(*hidden_bias)
print("Final output weights: ",end='')
print(*output_weights)
print("Final output bias: ",end='')
print(*output_bias)

print("\nOutput from neural network after 10,000 epochs: ",end='')
print(*predicted_output)
```



This lesson was developed by:

Robert Frans van der Willigen
CMD, Hogeschool Rotterdam
OKT 2020



http://empoweringthenatives.edublogs.org/2012/03/15/creative-commons-licenses/

http://creativecommons.org/licenses/


the functional art
alberto cairo
Christian Leborg
Visual Grammar
TECHNOLOGY TRENDS TO WATCH
the analytics revolution
Bill Franks
generative art
matt pearson
MADE to STICK
Heath & Dan Heath
EMPATHIC THINGS
Intieme technologie van wearables tot biohacking
INTERNET OF THINGS!
MAKERS
THE NEW INDUSTRIAL REVOLUTION
CHRIS ANDERSON
MARITIEME MAAKINDUSTRIE TERUG NAAR RDM
TV AND VIDEO
An analysis of evolving consumer habits
THE DISRUPTION MACHINE
Learning to Program with MATLAB
BUILDING GUI TOOLS
Port Management Program
R Data Analysis
10 TRENDS FOR 2014
A giant cage
Design for Information
HACKATHO
OPEN HAVE
1 VOLLE DAG
7 SEPTEMBER
WERELDHAVENDAGEN 2013
the myths of innovation
scott berkun
O'REILLY
resonate
Cool Infograph
Effective Communication with Data Visualization and Design
The VISU
DATA VISUALIZATION, BIG DATA, AND THE QUEST FOR BETTER DECISIONS
PHIL SIMON
OUR WORK HERE IS DONE
VISIONS OF A ROBOT ECONOMY
INVENT TO LEARN
Making, Tinkering, and Engineering in the Classroom
MORE SECRETS with Big Data Analytics
Processing
A Programming Handbook for Visual Designers and Artists
THE WALL STREET JOURNAL.
Guide to Information Graphics
Dona M. Wong
Customer Journey Mapping
Wearable Technology Design
The Hidden Pitfall in your Design
Making Things See
Learn by Discovery
TEACHING that STICKS
Chip Heath & Dan Heath
Protecting Data and Privacy in the Cloud
SOGETI
DISCOVER METEOR
Making Things Talk
The Internet of Things
Business Adapts to a New Style of Computer
EVOLUTION OF AGILE

Foundations of Machine Learning

Introduction to ML

Mehryar Mohri
Courant Institute and Google Research
mohri@cims.nyu.edu

HOGESCHOOL ROTTERDAM