

Machine learning introduction

Part I – From neurons to networks

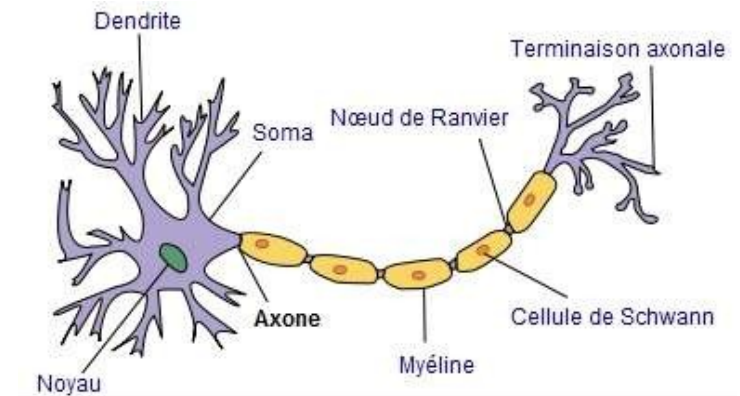
2 - The Formal Neuron

Simon Gay

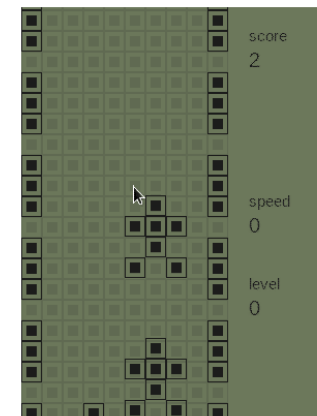
Part I-2 : the formal neuron

- **A simplified model of the neuron:**

- The biological neuron:
 - Dendrites collect input signals
 - The body processes these signals
 - An axon conveys the output signal
 - Information is encoded with impulses (frequency)



- The model must be simplified to work on a computer:
 - /\ The model of the formal neuron is a VERY simplified model of the biological neuron

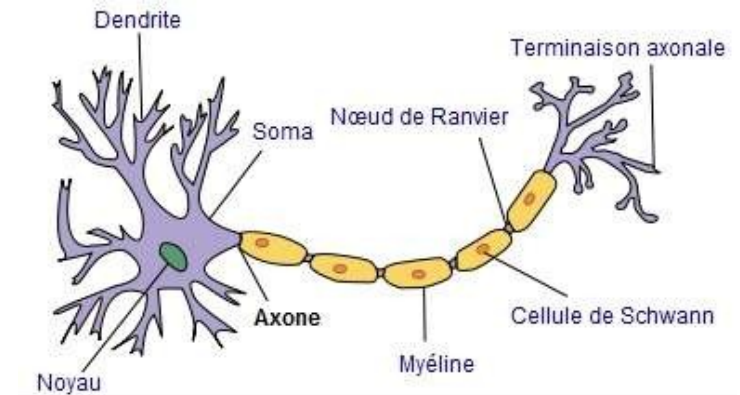


<https://github.com/tshiamobhuda/Racing>

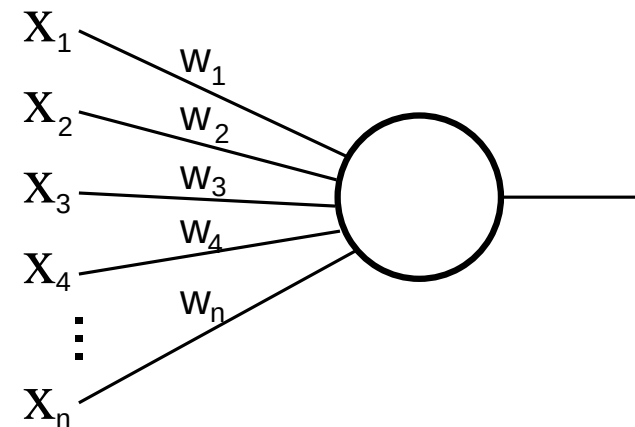
Part I-2 : the formal neuron

- **A simplified model of the neuron:**

- The biological neuron:
 - Dendrites collect input signals
 - The body processes these signals
 - An axon conveys the output signal
 - Information is encoded with impulses (frequency)



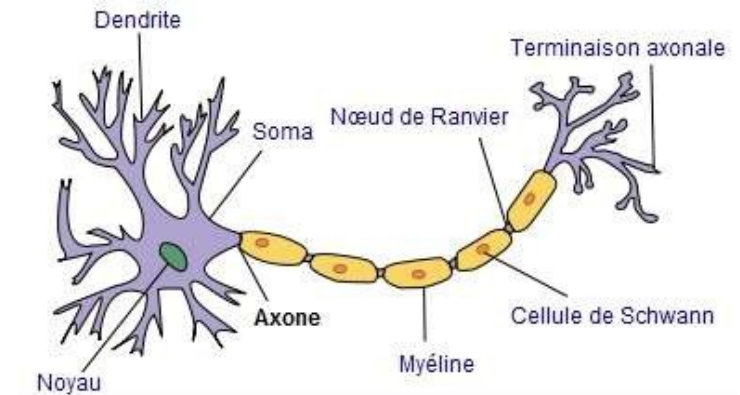
- The model must be simplified to work on a computer:
 - Frequency is replaced with a real value
 - Input: a vector of real values
 - Vector $X = \{x_1, x_2, x_3, \dots, x_n\}$
 - A vector of real values replace synapses
 - Vector $W = \{w_1, w_2, w_3, \dots, w_n\}$



Part I-2 : the formal neuron

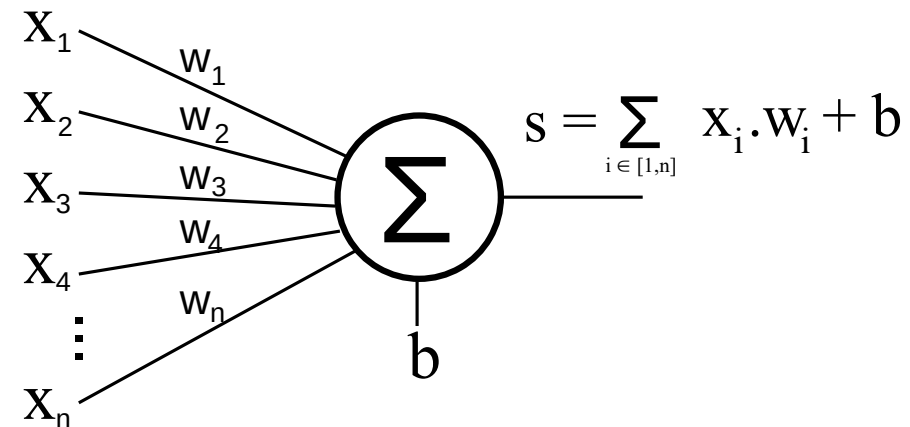
- **A simplified model of the neuron:**

- The biological neuron:
 - Dendrites collect input signals
 - The body processes these signals
 - An axon conveys the output signal
 - Information is encoded with impulses (frequency)



- The model must be simplified to work on a computer:

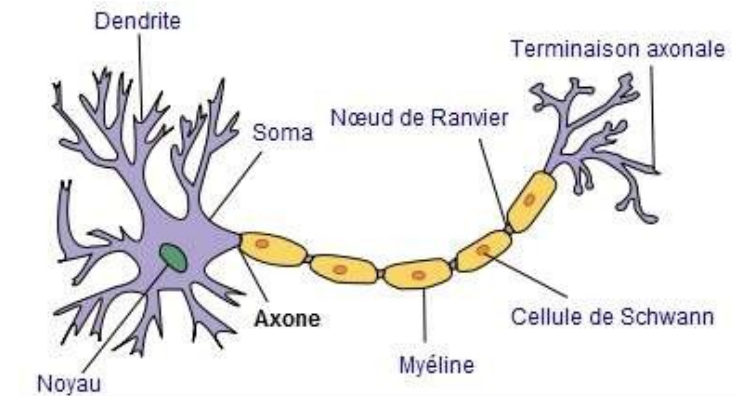
- Input processing:
 - Sum of inputs weighted by synaptic weights
 - A bias b



Part I-2 : the formal neuron

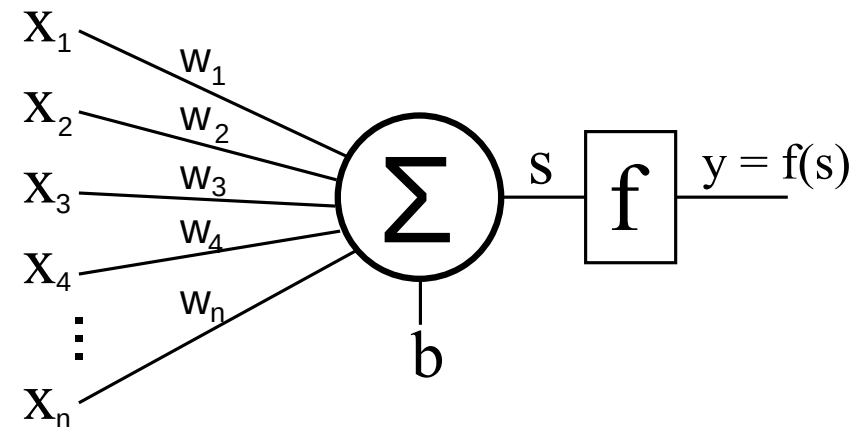
- **A simplified model of the neuron:**

- The biological neuron:
 - Dendrites collect input signals
 - The body processes these signals
 - An axon conveys the output signal
 - Information is encoded with impulses (frequency)



- The model must be simplified to work on a computer:

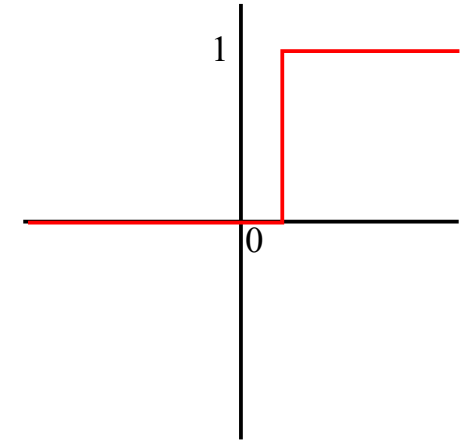
- The output passes through an activation function
 - Limits the neuron's output value



Part I-2 : the formal neuron

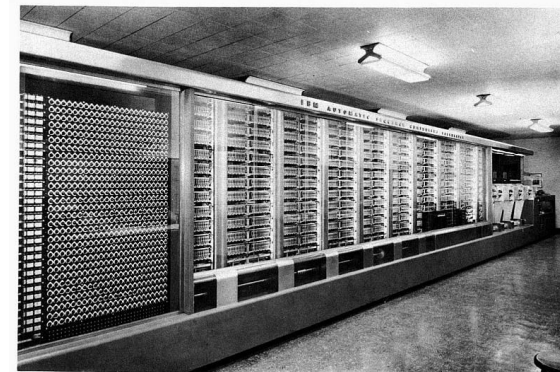
- The activation function

- Threshold function : $S = 1$ when $y > \text{threshold}$
 $S = 0$ otherwise



(note: The threshold value is not important due to the bias)

- Function used in the first neuron models (proposed by Warren McCulloch and Walter Pitts in **1943**)
- Several limitations:
 - Neurons' outputs cannot be compared
 - In a network, output “strength” cannot be transmitted to other neurons

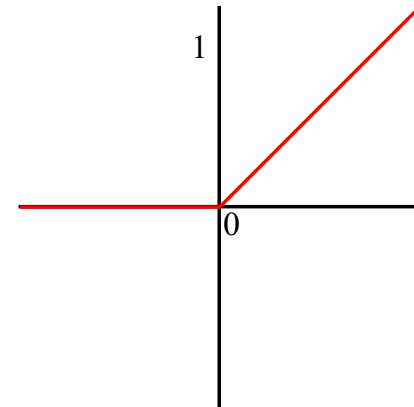
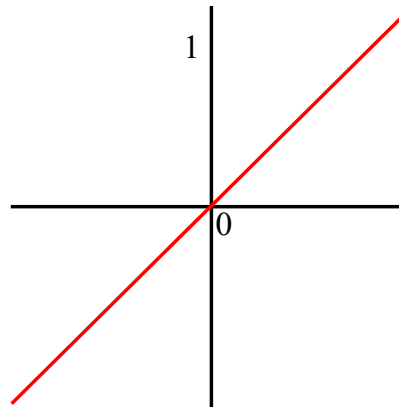


Part I-2 : the formal neuron

- The activation function

- Linear functions and ReLU (Rectified Linear Unit)

- $y = x$
- $y = \max(0, x)$



- Very simple functions
- ReLU allows removing negative values
 - Often used in deep learning approaches
- The output value can be greater than 1 (Weighted sum s must approach 1)
- Constant gradient (variations of a weight have the same effect on the output)

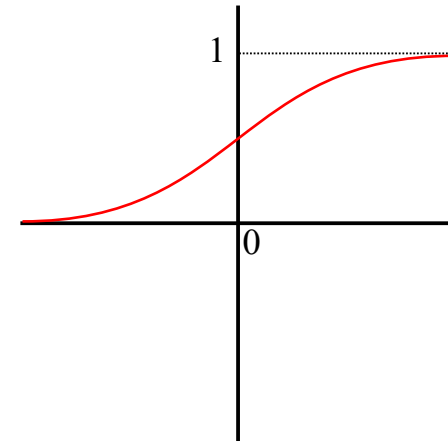
Part I-2 : the formal neuron

- **The activation function**

- Sigmoid function

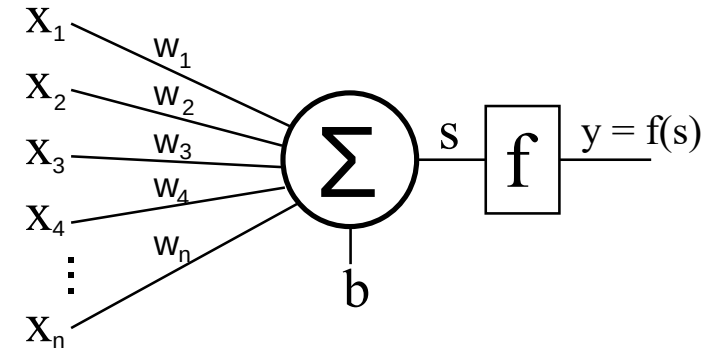
- $y = \frac{1}{1 + e^{-x}}$

(note: the tanh function is also used)



- More complex functions
- High values provide an output close to 1 (or 0 for negative values)
- The gradient tends to 0 for high values
 - Stabilization of weights (variations become weaker and weaker)
 - But we lose the gradient (can be problematic when using multiple layers)
- Another great advantage (we will see this later)

Part I-2 : the formal neuron



- **Methods for supervised learning**

- Hebb rule (1949) : “cells that fire together, wire together” (biological observations)
 - $w_i^{t+1} = w_i^t + \alpha \cdot x_i \cdot r$: if result r and input x_i have same sign, weight w_i is reinforced
 - α Is the *learning rate* (limiting weights variations)
- Delta rule : weights are modified according to the output error :

$$\Delta = r - y$$

- Perceptron rule (Rosenblatt, 1957) : use threshold activation function, thus $\Delta \in \{ -1 ; 0 ; 1 \}$
 - Widrow-Hoff rule (1960) : continuous activation functions, thus $\Delta \in \mathbb{R}$
- Weights are modified to makes the output closer from expected result
- principles: modify each weight w_i according to :
 - The input value (a higher value will have a greater influence on the output)
 - The difference (Δ) between output and expected result (the greater the difference, the more the weights must be modified)
 - $w_i^{t+1} = w_i^t + \alpha \cdot x_i \cdot \Delta$
- Neurons are trained on large number of examples

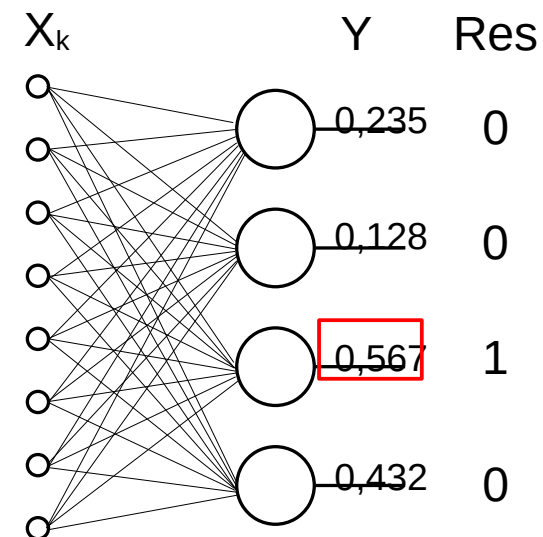
Part I-2 : the formal neuron

- ...And some for unsupervised learning

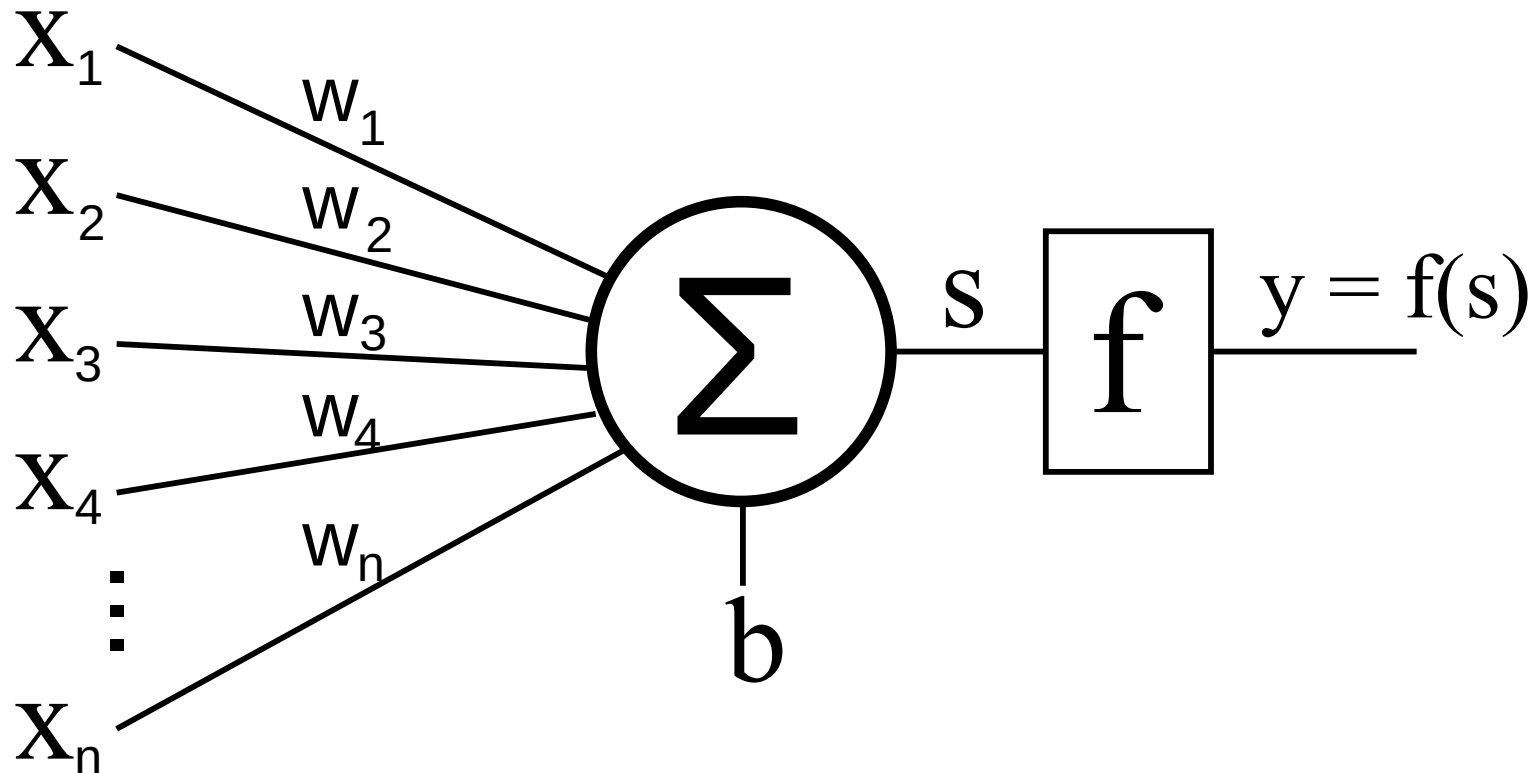
- Winner-Takes-All :
 - Multiple neurons in competition
 - The neuron with highest input inhibates others
 - Weights are modified until convergence
- Grossberg rule :
 - Neuron with highest output update its weights to get closer to the input vector

$$w_i^{t+1} = w_i^t + \alpha \times (x_i - w_i^t)$$

- Progressively, the group of neurons define classes of input vectors



Part I-2 : the formal neuron

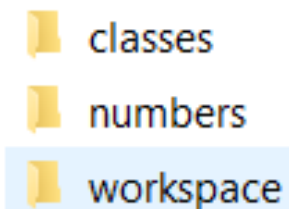


Enough with theory, now to practice!

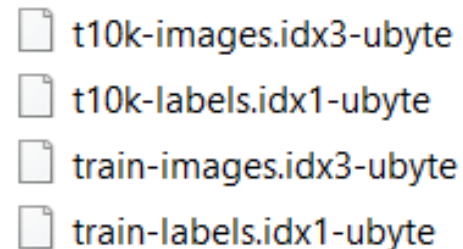
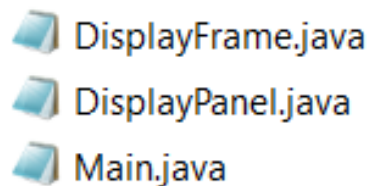
Part I-2 : the formal neuron

- **The Workspace**

- Create a '*Workspace*' folder somewhere (e.g. in 'my documents' or 'Desktop')
- Create a folder 'classes' and a folder 'numbers' beside your *workspace* folder



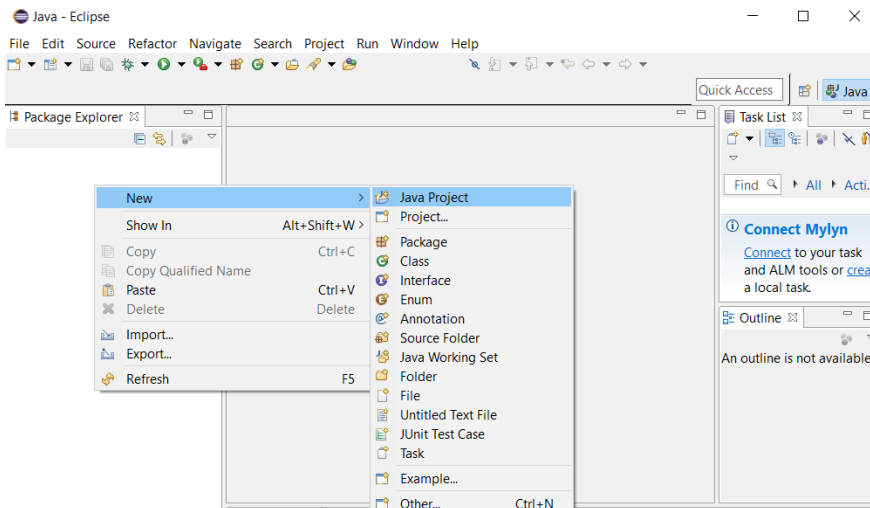
- Download the 3 files at <https://github.com/gaysimon/ARTISAN2022/tree/main/classes>
 - Click on green button 'Code' then 'Download Zip'
 - Unzip the three files in 'classes' folder
- Download the 4 dataset files at <http://yann.lecun.com/exdb/mnist/> and unzip them in 'numbers' (!\ be sure that the four files have the same name as below)



Part I-2 : the formal neuron

- **The Workspace**

- open Eclipse and select your 'workspace' folder, then click on 'Workbench' (top right corner)
- Create a new project 'Neuron'
 - Right click in the package explorer (left part) → new → Java project
 - Name your project 'Neuron', then click 'Finish'



Create a Java Project

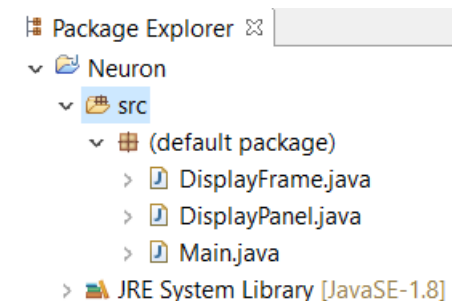
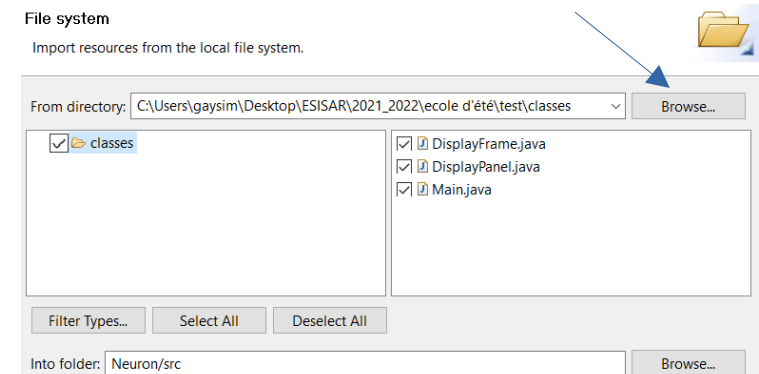
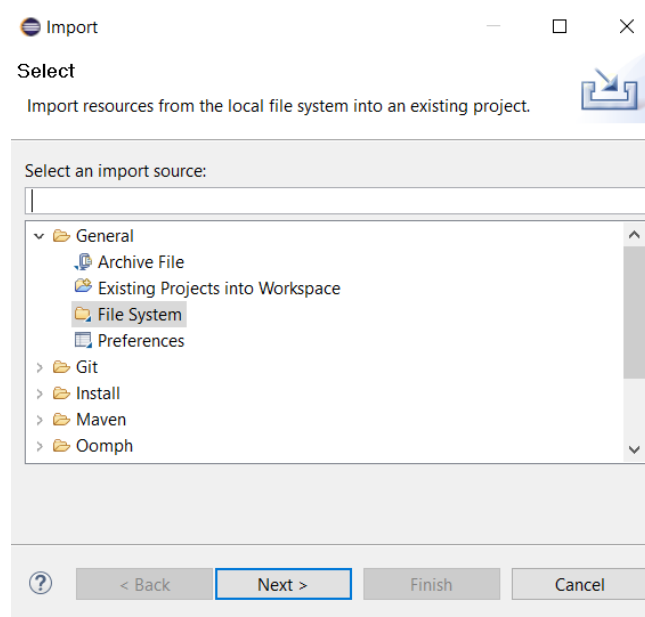
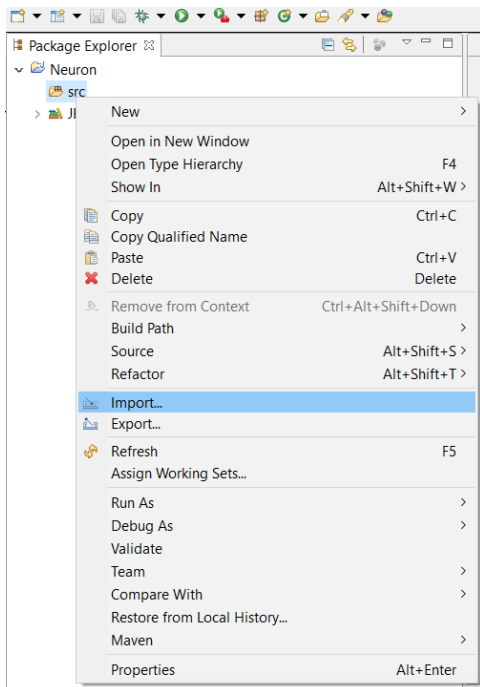
Create a Java project in the workspace or in an external location.

A screenshot of the 'Create a Java Project' dialog box. The 'Project name' field is filled with 'Neuron'. The 'Use default location' checkbox is checked. The 'Location' field shows the path 'C:\Users\gaysim\Desktop\ESISAR\2021_2022\ecole d'été\test\workspace\'. The 'JRE' section has three options: 'Use an execution environment JRE:' (selected) with 'JavaSE-1.8' in the dropdown, 'Use a project specific JRE:' with 'jre1.8.0_211' in the dropdown, and 'Use default JRE (currently 'jre1.8.0_211')'. A 'Configure JREs...' link is at the bottom right.

Part I-2 : the formal neuron

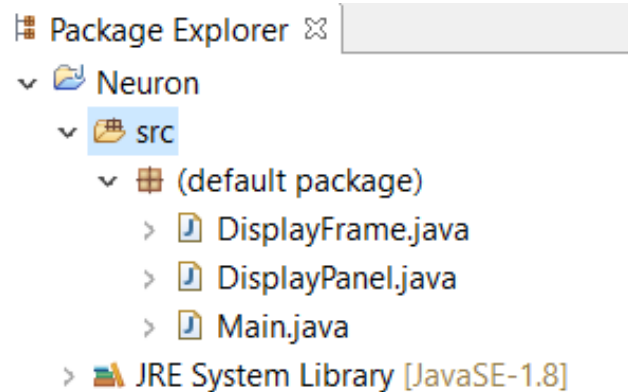
• The Workspace

- Import the java file from the previously downloaded archive
 - Expand the project in package explorer
 - Right click on 'src' (left part) → import, then 'general' → 'file system'.
 - Browse to your 'classes' folder and select *Main.java*, *DisplayFrame.java* and *DisplayPanel.java*
 - Click on 'Finish' to complete the import



Part I-2 : the formal neuron

- The Workspace



- 3 classes :
 - **Main.java** :
 - main function
 - A function to load datasets (*setData*)
 - Bases for training and exploiting a neuron
 - **DisplayFrame.java** and **DisplayPanel.java** :
 - Create a window to display properties of the neuron

Part I-2 : the formal neuron

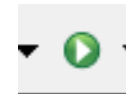
- The Workspace

- Finally, double-click on the *Main* class to open it, and change the *path* value (line 9) to indicate the position of your 'numbers' folder.

- *//\ use '\\' folder separator on windows and '/' on Linux/Mac*

```
Main.java x
1 import java.io.BufferedReader;
5
6
7 public class Main {
8
9     public static String PATH="C:\\Users\\gaysim\\Desktop\\numbers\\"; // path to images
10
11     public static int size_x=28; // size of the image (set when image is loaded)
12     public static int size_y=28;
```

- Test the application with 'run' button to test your installation



Display

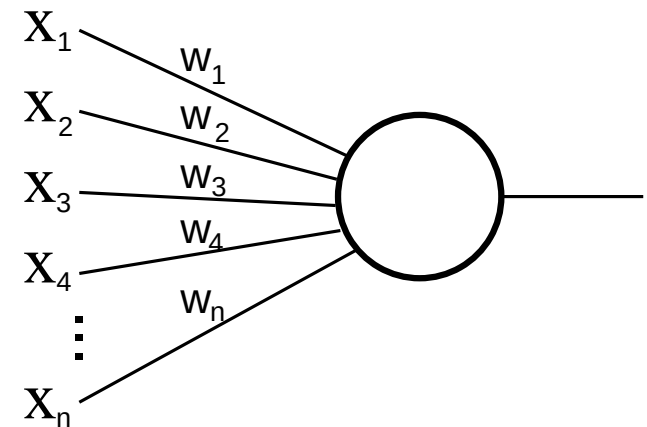


Part I-2 : the formal neuron

- **Creating an artificial neuron**

- We will implement an artificial neuron and add it to our project
- First, create a new class in your project
 - Right click on 'default package' → new class. Name your class 'Neuron'
- We first implement weights and a bias

```
1 public class Neuron {  
2  
3     public float[] synaps;  
4     public float bias;  
5  
6  
7 }  
8  
9
```

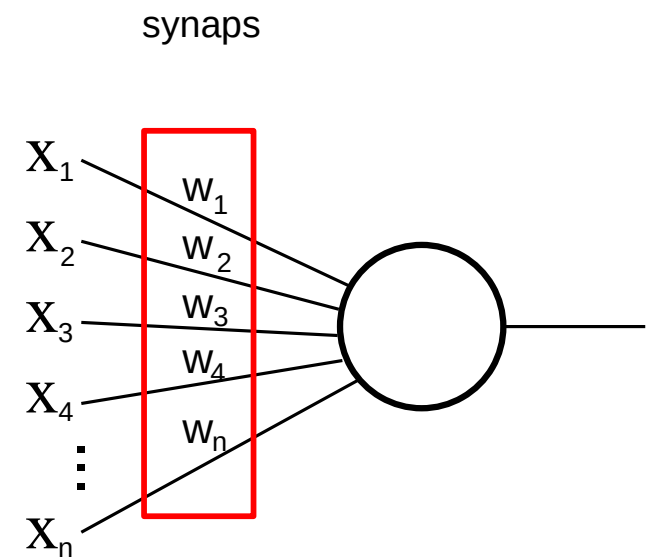


Part I-2 : the formal neuron

- **Creating an artificial neuron**

- We then write the class constructor, and initialize the weight vector and bias
 - The parameter *size* gives the number of weights

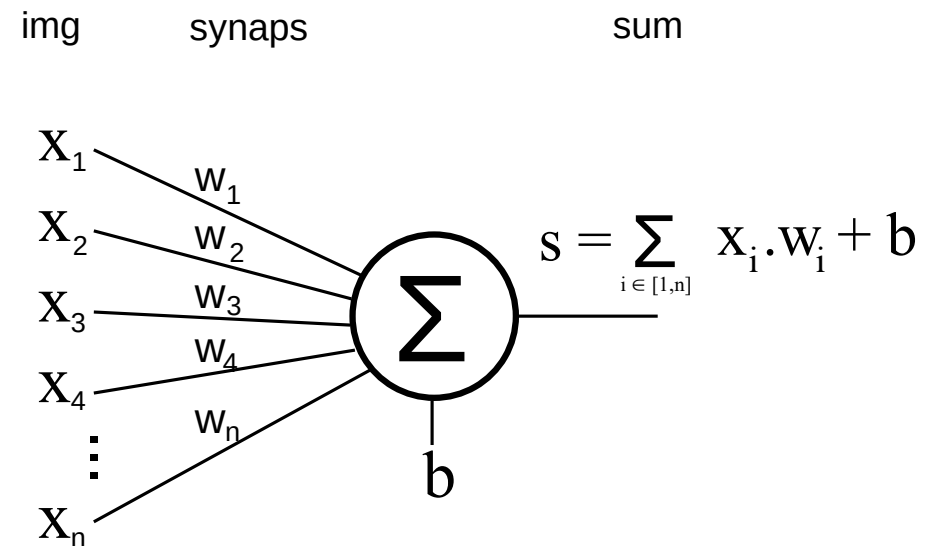
```
1  
2 public class Neuron {  
3  
4     public float[] synaps;  
5     public float bias=0; ←  
6  
7     public Neuron(int size){  
8         synaps=new float[size];  
9     }  
10 }  
11
```



Part I-2 : the formal neuron

- **Creating an artificial neuron**
 - We add a function to compute the output of the neuron
 - A vector *img* is used as parameter (input vector)

```
1  Main.java  Neuron.java  ✕
2  public class Neuron {
3
4      public float[] synaps;
5      public float bias=0;
6
7      public Neuron(int size){
8          synaps=new float[size];
9      }
10
11
12      public float compute(float[] img){
13
14          // compute value
15          float sum=0;
16          for (int i=0;i<synaps.length;i++){
17              sum+=img[i]*synaps[i];
18          }
19          sum+=bias;
20
21          return sum;
22      }
23 }
```



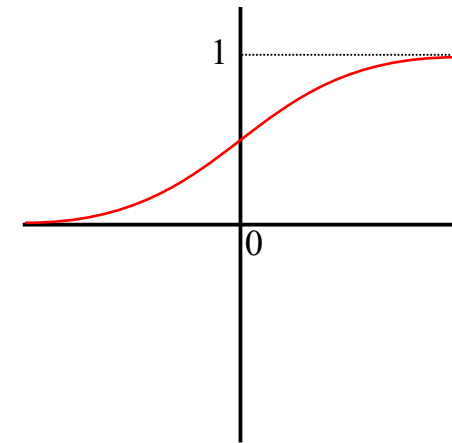
Part I-2 : the formal neuron

- **Creating an artificial neuron**

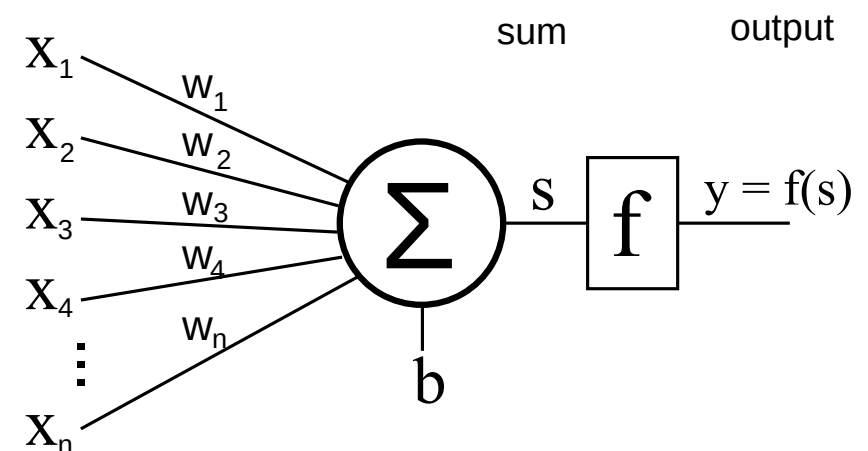
- We forgot the activation function !

```
2 public class Neuron {
3
4     public float[] synaps;
5     public float bias=0;
6
7     public float output=0;
8
9     public Neuron(int size){
10         synaps=new float[size];
11     }
12
13     public float compute(float[] img){
14
15         // compute value
16         float sum=0;
17         for (int i=0;i<synaps.length;i++){
18             sum+=img[i]*synaps[i];
19         }
20         sum+=bias;
21
22         output=activation(sum);
23         return output;
24     }
25
26     private float activation(float x){
27         return (float) (1 / (1+Math.exp(-x)));
28     }
29 }
```

We will also
record the
output value



$$y = \frac{1}{1 + e^{-x}}$$



Part I-2 : the formal neuron

- **Creating an artificial neuron**

- We then define a function to train the neuron :

- Widrow-Hoff :

- $\Delta = r - y$ → difference between expected result and output
 - $w_i^{t+1} = w_i^t + \alpha \cdot x_i \cdot \Delta$ → each weight is updated in proportion to related input value

- First, we add the delta value and a learning rate with a value of 0,01

- Then, a learning function that computes the delta value, using input vector and expected result

```
Main.java  *Neuron.java ✕
1
2 public class Neuron {
3
4 → public float learnRate=0.01f;
5
6     public float[] synaps;
7     public float bias=0;
8
9     public float output=0;
10 → public float delta=0;
11
12 public Neuron(int size){
```

```
25         output=activation(sum);
26         return output;
27     }
28
29 → public void learn(float[] img, int res){
30     delta=res-output;
31
32 }
33
34 → private float activation(float x){
```

Part I-2 : the formal neuron

- **Creating an artificial neuron**

- We then define a function to train the neuron :

- Widrow-Hoff :

- $\Delta = r - y$ → difference between expected result and output
 - $w_i^{t+1} = w_i^t + \alpha \cdot x_i \cdot \Delta$ → each weight is updated in proportion to related input value

- Finally, the learn function update weight according to Widrow-Hoff rule :

- Don't forget the bias !

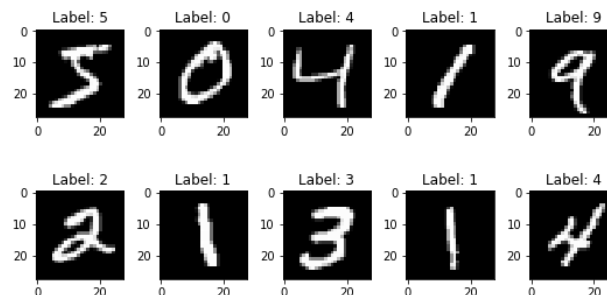
```
29 public void learn(float[] img, int res){  
30     delta=res-output;  
31  
32     for (int i=0;i<synaps.length;i++){  
33         synaps[i]+= learnRate * delta * img[i];  
34     }  
35     bias+=learnRate * delta;  
36 }
```





- **Congratulations ! Your neuron is complete !**

Part I-2 : the formal neuron

- **Training and exploiting a neuron**

- We will use our neuron on a dataset called MNIST number.
 - This dataset is composed of two sets of small images of 28x28 pixels of handwritten digits (0 to 9)



-  `t10k-images.idx3-ubyte`
-  `t10k-labels.idx1-ubyte`
-  `train-images.idx3-ubyte`
-  `train-labels.idx1-ubyte`

- A first dataset of 60 000 samples is used for training
- A second dataset of 10 000 samples is used for testing
- Each dataset is composed of two files : a file containing the pixel values of images, and a file containing the label (number) of images
- A function is provided in *main* class to read these files and load samples in a matrix and labels in a vector (*setData*)

Part I-2 : the formal neuron

- Training and exploiting a neuron

- In class *main*, uncomment the line adding the neuron as a variable

```
29  
30 public Neuron neuron; // neuron  
31  
32
```

- The neuron must have a weight for each pixel of image. We thus initialize the neuron with 28x28 weights :

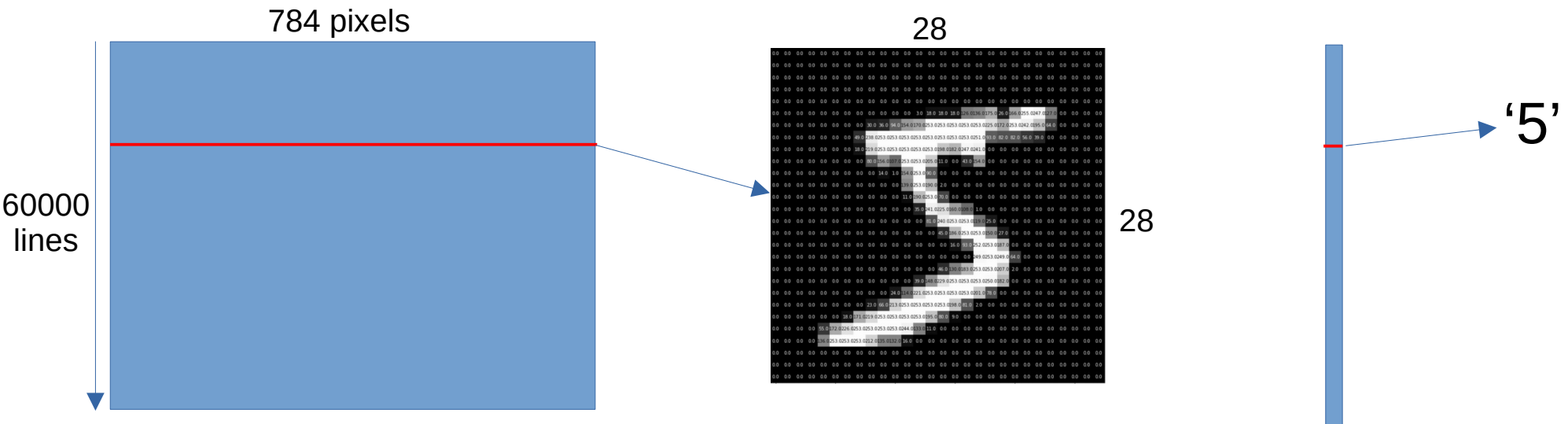
```
34 ///////////////////////////////////////////////////  
35 public Main() {  
36  
37 ///////////////////////////////////////////////////  
38 // initialization  
39 ///////////////////////////////////////////////////  
40 neuron=new Neuron(size_x*size_y); ← 784 inputs  
41  
42 // load test matrix  
43 setData("train-images.idx3-ubyte","train-labels.idx1-ubyte");  
44  
45 // initialize display frame  
46 display=new DisplayFrame(this);  
47
```

- The lines after load the training dataset and initialize the display window

Part I-2 : the formal neuron

- Training and exploiting a neuron

- The MNIST training dataset is loaded in a matrix (for the images) and a vector (for the labels)



- We will get sample images by reading the matrix line by line and the label by reading the same line on label vector
 - The neuron does not consider the position of pixels on the 2D image : it gets the image as a vector of pixels !

Part I-2 : the formal neuron

- **Training and exploiting a neuron**

- We will train our neuron to recognize a digit. Choose a digit (0-9) for the number variable (you can change later)

```
22  
23     public int number=2;    // choose a number  
24
```

- The training cycle works as follow :
 - For each sample of the dataset
 - Get next sample and its label
 - Compute output value of the neuron
 - Reinforce the neuron using expected result (delta)
- The neuron can be trained on the dataset multiple times to improve results. Each dataset cycle is called 'epoch '
 - We will train the neuron on 50 epochs

```
// apply the learning process 50 times  
for (epoch=0;epoch<50;epoch++){  
  
    float sumdelta=0;    // measure total error of the current epoch  
  
    // for each test image  
    for (test=0;test<matrixImages.length;test++){  
  
        ///////////  
        // TODO //  
        ///////////  
  
        display.repaint();  
  
        try {Thread.sleep(100);  
        } catch (InterruptedException e) {e.printStackTrace();}  
    }  
  
    // display mean error of the epoch  
    System.out.println("epoch n°"+epoch+" : "+(sumdelta/nbTests));  
}  
  
System.out.println("learning completed");
```

Part I-2 : the formal neuron

- Training and exploiting a neuron

- First, we define the expected result : 1 when the digit is the right one, 0 otherwise

```
//////////////////////////////////////////  
// learning  
//////////////////////////////////////////  
  
// apply the learning process 50 times  
for (epoch=0;epoch<50;epoch++){  
  
    float sumdelta=0;    // measure total error of the current epoch  
  
    // for each test image  
    for (test=0;test<matrixImages.length;test++){  
  
        // set output value  
        int expected=0;  
        if (matrixLabels[test] == number) expected=1;  
  
        display.repaint();  
  
        try {Thread.sleep(100);  
        } catch (InterruptedException e) {e.printStackTrace();}  
    }  
}
```

Part I-2 : the formal neuron

- Training and exploiting a neuron

- Then, the neuron's output is computed on sample image(and stored in its output variable)

```
// apply the learning process 50 times
for (epoch=0;epoch<50;epoch++){

    float sumdelta=0;    // measure total error of the current epoch

    // for each test image
    for (test=0;test<matrixImages.length;test++){

        // set output value
        int expected=0;
        if (matrixLabels[test] == number) expected=1;

        // process neuron
        neuron.compute(matrixImages[test]); // get result

        display.repaint();

        try {Thread.sleep(100);}
        catch (InterruptedException e) {e.printStackTrace();}
    }
}
```

Part I-2 : the formal neuron

- Training and exploiting a neuron

- Finally, reinforce the neuron. We also record the delta to observe the evolution of predictions

```
// apply the learning process 50 times
for (epoch=0;epoch<50;epoch++){

    float sumdelta=0;    // measure total error of the current epoch

    // for each test image
    for (test=0;test<matrixImages.length;test++){

        // set output value
        int expected=0;
        if (matrixLabels[test] == number) expected=1;

        // process neuron
        neuron.compute(matrixImages[test]); // get result

        // reinforce neuron
        neuron.learn(matrixImages[test], expected);

        // add delta to the error sum
        sumdelta+=Math.abs(neuron.delta);

        display.repaint();


        try {Thread.sleep(100);}
        catch (InterruptedException e) {e.printStackTrace();}
    }
}
```

Part I-2 : the formal neuron

- Training and exploiting a neuron

- In class DisplayPanel, uncomment the second part in PaintComponent procedure (remove '/*' on line 34) to display the neuron's weights. Then, test your application

```
30         g.fillRect(10+5*i, 10+5*j, 5, 5);
31     }
32 }
33
34     for (int i=0;i<Main.size_x;i++){
35         for (int j=0;j<Main.size_y;j++){
36             val=(int) (main.neuron.synaps[i+Main.size_x*j]*50)+128;
37             if (val<0) val=0;
38             if (val>255) val=255;
39             g.setColor(new Color(val,val,val));
40             g.fillRect(180+3*i, 10+3*j, 3, 3);
41         }
42     }/**/
43
44     /*for (int n=0;n<10;n++){
45         for (int i=0;i<Main.size_x;i++){
```



- If the program runs correctly, comment or remove the sleep fonction in main class

```
display.repaint();

//try {Thread.sleep(100);
//} catch (InterruptedException e) {e.printStackTrace();}

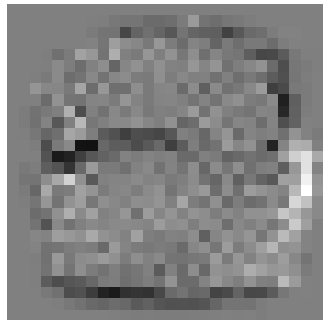
}
```

- What do you observe on neuron's weights and average delta ?

Part I-2 : the formal neuron

- **Training and exploiting a neuron**

- The weights form the 'average' shape of the selected digit
- The average delta first decreases quickly but seems to converge



- Now, let's test our neuron on the test dataset (loaded hereafter)

```
88 ///////////////////////////////////////////////////  
89 // exploitation  
90 ///////////////////////////////////////////////////  
91  
92 // test neuron on test dataset  
93 int errors=0;  
94  
95 // load test matrix  
96 setData("t10k-images.idx3-ubyte","t10k-labels.idx1-ubyte");  
97
```

Part I-2 : the formal neuron

- **Training and exploiting a neuron**

- We simply get neuron's predictions for each sample image :
- We can also count the number of prediction errors :

```
// test each image of the
for (test=0;test<matrixImages.length;test++){

    // process neuron
    float res=neuron.compute(matrixImages[test]);

    if (res<0.001) res=0;    // avoid low values displayed as "9.9586 x 10^-5"
    System.out.println(matrixLabels[test]+" -> "+res);

    // count errors
    if (matrixLabels[test]==number && res<0.5 || matrixLabels[test]!=number && res>0.5) errors++;

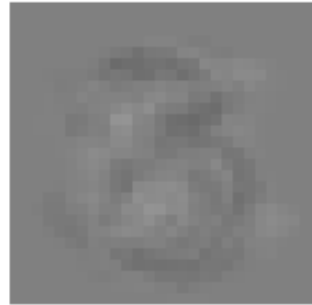
}
```

- Not bad for 10 000 samples errors : 254

Part I-2 : the formal neuron

- **The formal neuron**

- The formal neuron makes an average of positive and negative results to generate an “average image”, exposing the features of the element to recognize



- This average image allows detecting the presence of the associated element on an unknown input vector.
- **But Why does this works ?**

Part I-2 : the formal neuron

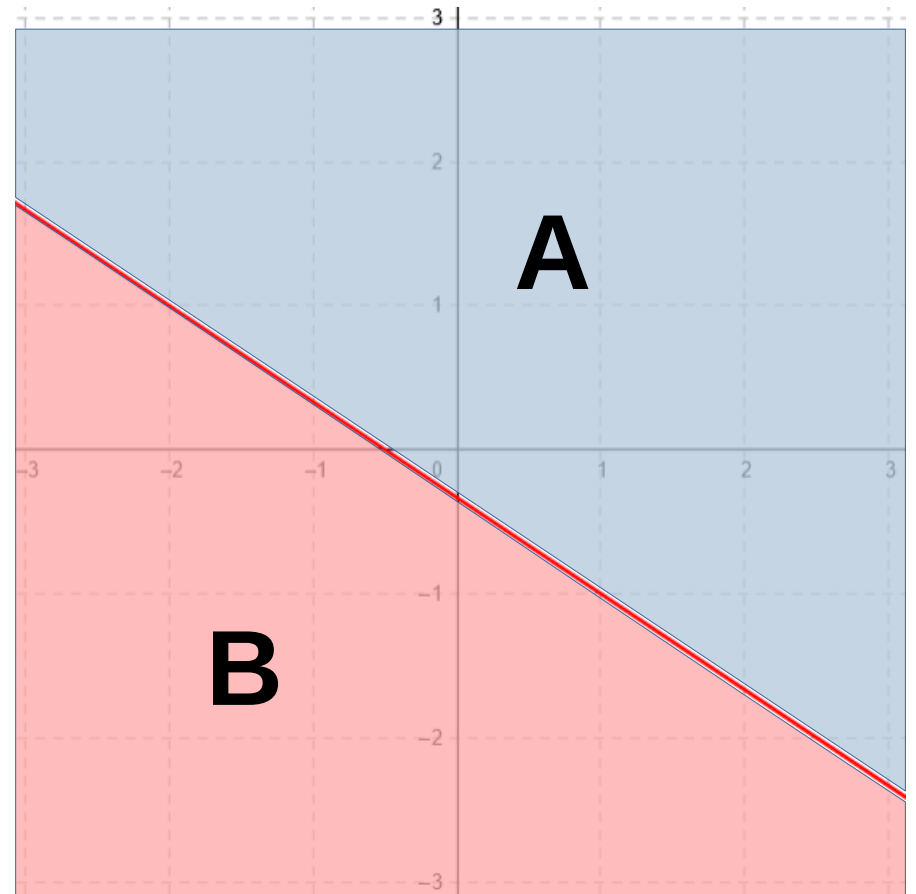
- Let's consider a linear function

example :

$$3.y + 2.x + 1 = 0 \quad (y = -(2/3).x - 1/3)$$

This function separates the plane in 2:

- $3.y + 2.x + 1 > 0$
→ area A above the line
- $3.y + 2.x + 1 < 0$
→ area B below the line



Part I-2 : the formal neuron

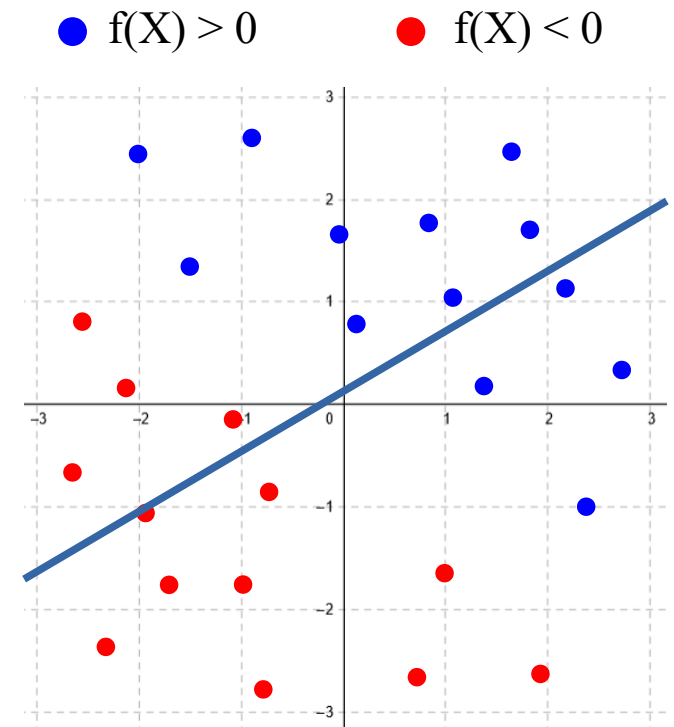
- **And if we do not know the function parameters ?**

We start with a function $a.x + b.y + c = 0$ with unknown parameters (a,b,c)

We have a set of points $X_i = [x_i, y_i]$ which position $f(X_i) > 0$ or $f(X_i) < 0$ is known

→ We have to find a triplet (a, b, c) satisfying all point conditions

- We get a random triplet
- Each point is tested
 - If $f(X_i) > 0$ but $ax_i + by_i + c < 0$
 - $ax_i + by_i + c$ must be increased
 - If $f(X_i) < 0$ but $ax_i + by_i + c > 0$
 - $ax_i + by_i + c$ must be decreased
 - otherwise, we do not change parameters



Part I-2 : the formal neuron

- **And if we do not know the function parameters ?**

For each parameter a , b and c , we slightly increase or decrease the value proportionally to, respectively, x (a), y (b) and 1 (c)

$$a \leftarrow a + \alpha \cdot x_i \quad \text{or} \quad a \leftarrow a - \alpha \cdot x_i$$

$$b \leftarrow b + \alpha \cdot y_i \quad \text{or} \quad b \leftarrow b - \alpha \cdot y_i$$

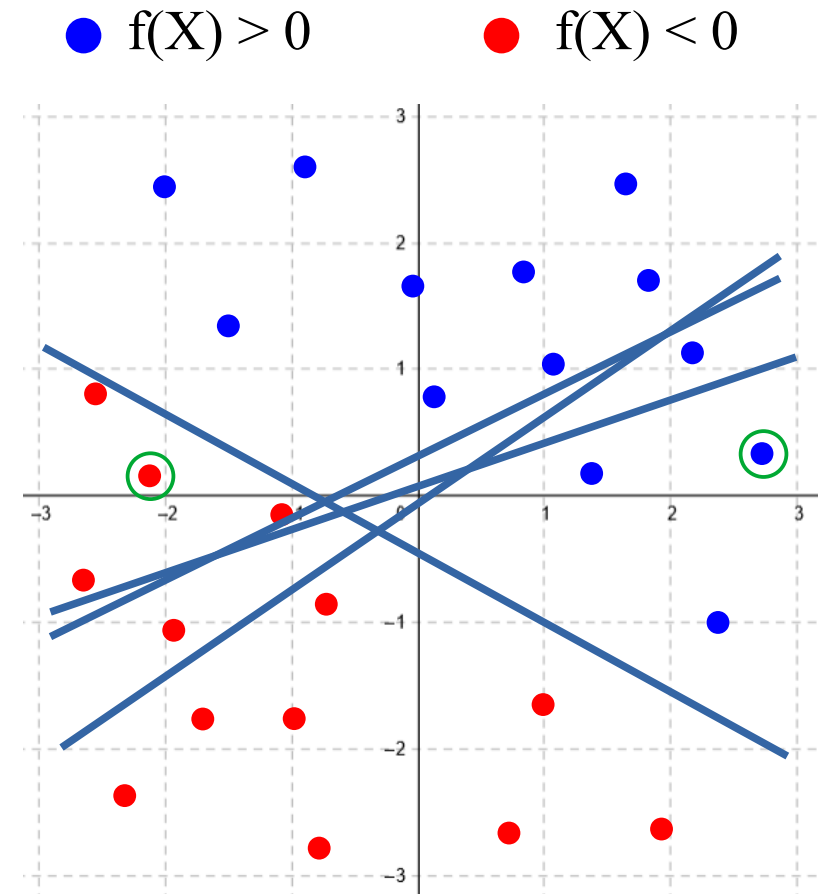
$$c \leftarrow c + \alpha \cdot 1 \quad \text{or} \quad c \leftarrow c - \alpha \cdot 1$$

- **If we note $r_i = 1$ when $f(X_i) > 0$ and $r_i = -1$ when $f(X_i) < 0$**
 - then
 - $a \leftarrow a + \alpha \cdot r_i \cdot x_i$
 - $b \leftarrow b + \alpha \cdot r_i \cdot y_i$
 - $c \leftarrow c + \alpha \cdot r_i$
 - Points X_i are tested, until they are all on the right side of the line

Part I-2 : the formal neuron

- The parameters will converge until finding a solution

- Algorithm :
errors = **true**
while errors **do**
 errors = **false**
 for each X_i **do**
 if $f(X_i) \cdot (a \cdot x_i + b \cdot y_i + c) < 0$ **do**
 $a += \alpha \cdot r_i \cdot x_i$
 $b += \alpha \cdot r_i \cdot y_i$
 $c += \alpha \cdot r_i$
 errors = **true**
 end if
 end for each
end while



Part I-2 : the formal neuron

- What has this got to do with the neuron ?

- Let's write:

$$- a \cdot x + b \cdot y + c = 0 \quad \rightarrow \quad w_1 \cdot x_1 + w_2 \cdot x_2 + b = 0$$

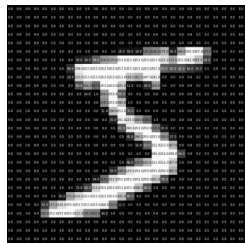
- We generalize to a space with n-dimensions:

$$\sum_k w_k \cdot x_k + b = 0$$

- Weights of a neuron form the equation of a hyperplane
- The reinforcement modifies weights to separate space in two groups of points

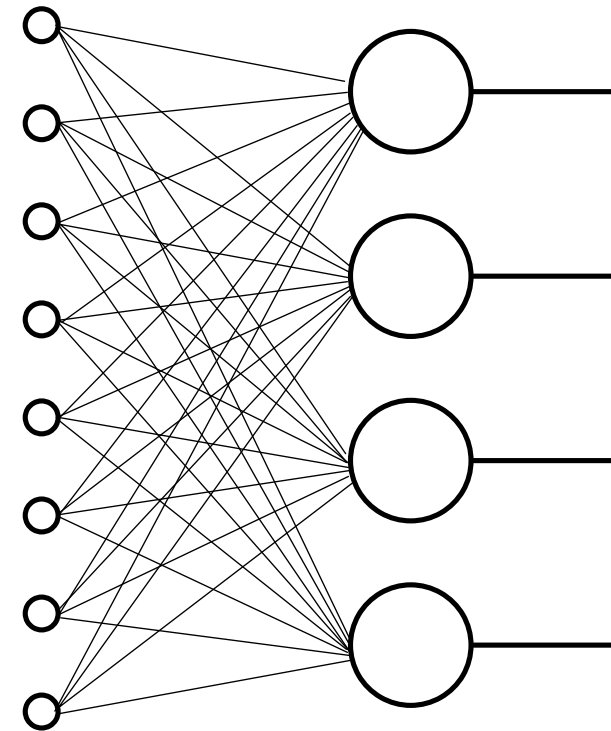
Part I-2 : the formal neuron

- A neuron can only recognize one type of element
- But with multiple neurons, it is possible to categorize multiple elements
- With a threshold activation function :
 - The output is a binary vector
 - Conversion to binary code, ASCII code...



{ 0 , 1 , 0 , 1 }

→ **Perceptrons**



Part I-2 : the formal neuron

- **Neurons with continual activation function (linear, ReLU, Sigmoid...)**
- **Each neuron is trained to recognize a category of element (e.g. a number)**
 - The training process must convert labels into binary vectors
 - '2' \rightarrow {0,0,1,0,0,0,0,0,0,0}
 - Each neuron needs a binary output for training
 - Neurons are in competition : the neuron with the greatest output provide the output of the network, with the confidence of the result



(0: 0.21) (1: 0,01) (2: 0,65) (3: 0,55) (4: 0,35) (5: 0,11) (6: 0,09)...

- **Often used for object recognition**

Part I-2 : the formal neuron

- **Implementing a layer of neuron**

- Simple way : create a class that manage a vector of neurons and provide a vector of results as output

```
2 public class Layer {
3
4     public Neuron[] layer;
5     public float[] output;
6
7     public float delta=0;
8
9     public Layer(int nb, int size){
10
11         layer=new Neuron[nb];    // declare neuron vector
12         for (int i=0;i<nb;i++){ // initialize each neuron
13             layer[i]=new Neuron(size);
14         }
15
16         output=new float[nb];    // declare the output vector
17     }
18 }
```

Initialize the vector of neurons

```
18
19 public float[] compute(float[] img){
20
21     for (int i=0;i<layer.length;i++){
22         output[i]=layer[i].compute(img);
23     }
24
25     return output;
26 }
```

Compute output of each neuron
And write their results in output vector

```
29 public void learn(float[] img, int[] results){
30
31     delta=0;
32
33     for (int i=0;i<layer.length;i++){
34         layer[i].learn(img, results[i]);
35
36         delta+=Math.abs(layer[i].delta);
37     }
38 }
```

Reinforce neurons individually
Notice that result is a vector (result for each neuron)

Part I-2 : the formal neuron

- **Implementing a layer of neuron**

- Training : the result vector must be obtained from labels :

```
58         // for each test image
59         for (test=0;test<matrixImages.length;test++){
60
61             // set output value
62             int[] expected=new int[10];
63             expected[matrixLabels[test]] = 1;
```

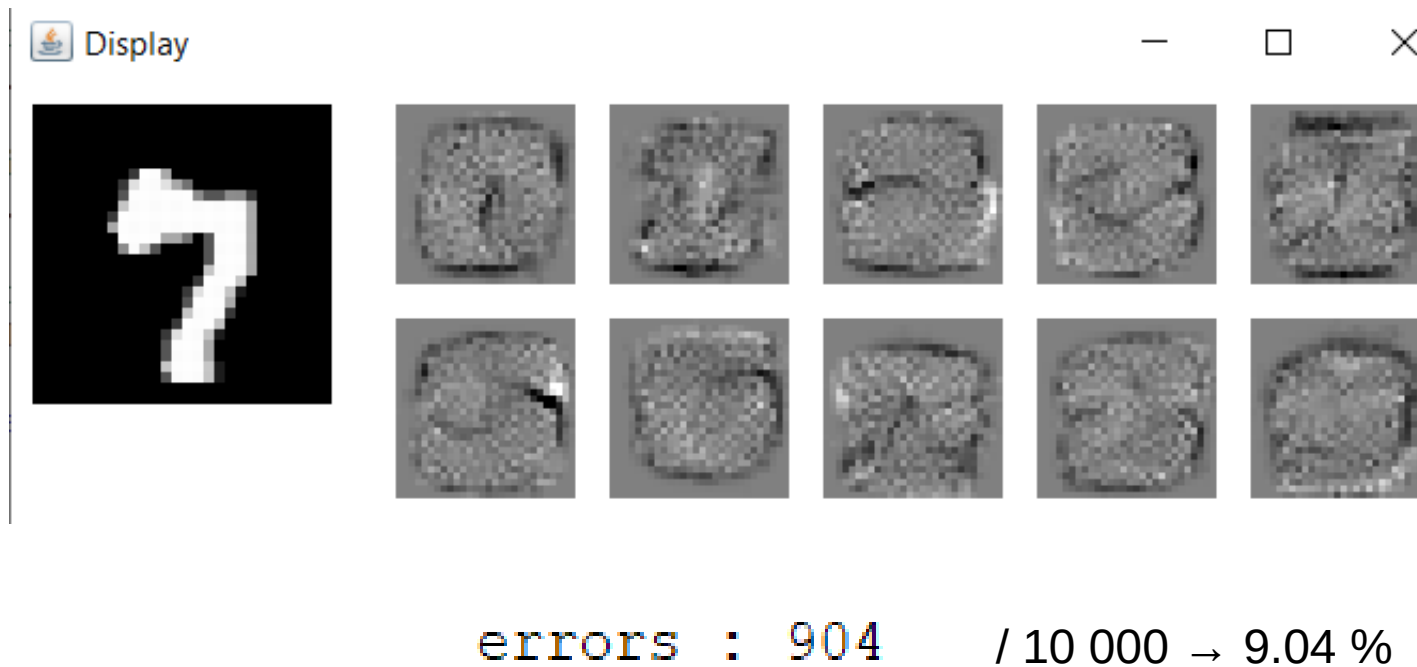
Expected is a vector of 10 values, with only one value at 1

- The result is obtained by finding the maximum output :

```
99         // test each image of the dataset
100        for (test=0;test<matrixImages.length;test++){
101
102            // process neurons
103            layer.compute(matrixImages[test]);
104
105            float max=0;
106            int imax=0;
107            for (int i=0;i<10;i++){
108                if (layer.output[i]>max){
109                    max=layer.output[i];
110                    imax=i;
111                }
112            }
113
114            System.out.println(matrixLabels[test]+" -> "+imax);
```

Part I-2 : the formal neuron

- Each neuron specializes for a digit, the highest output provide the number



ANNEX 1 provide details of this implementation