# Reconnaissance optique de caractères

- Approche connexioniste
- Application en C++
- Limitation aux chiffres manuscrits

# Plan :

**I. Pré-traitement**

a) Binarisation

b) Squelettisation

c) Segmentation

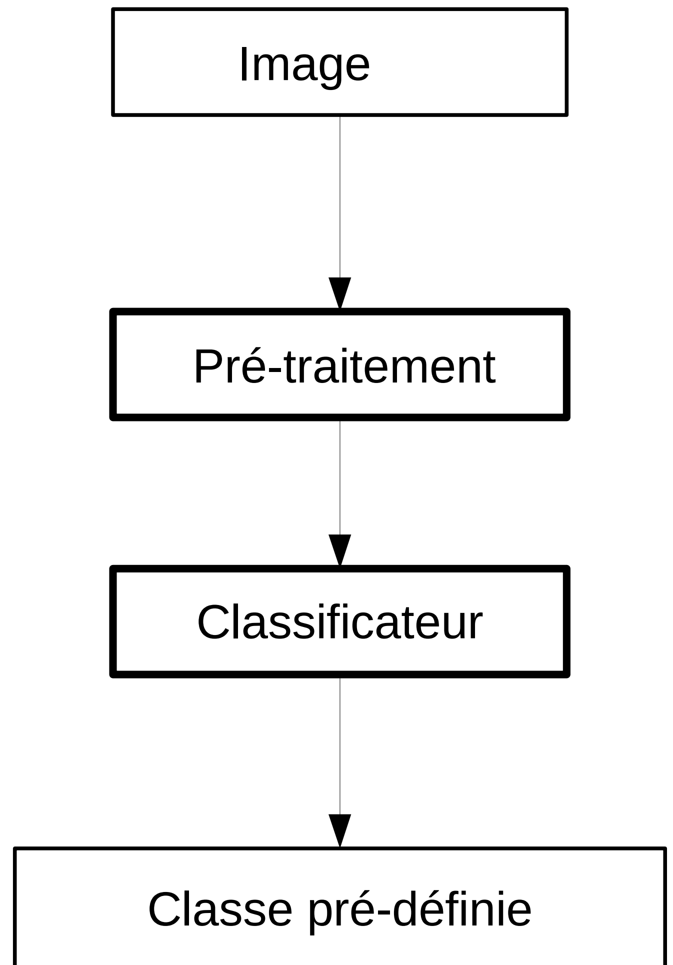
**II. Classification**

a) Structure

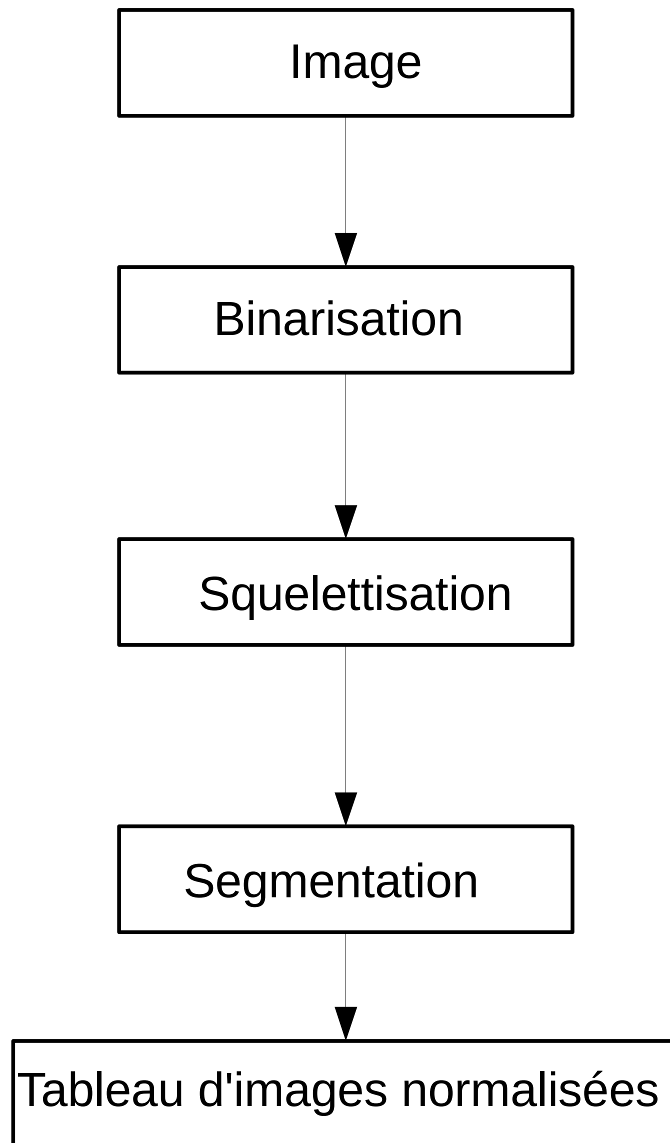
b) Apprentissage

c) Résultats

**III. Conclusion**



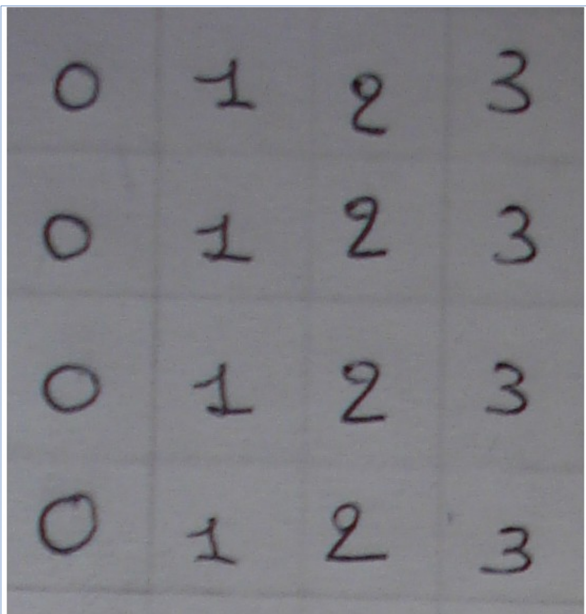
# I Pré-traitement



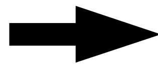
# I.1 Binarisation

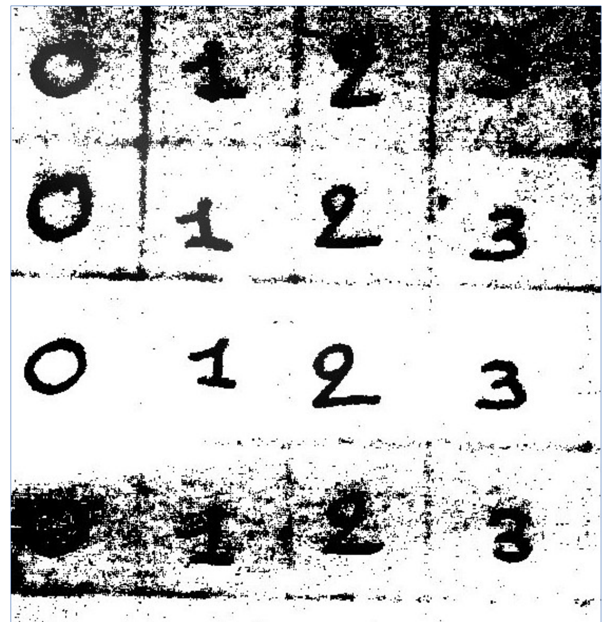
## a) Seuillage global :

- $S \in [\![0;255]\!]$



Selection(570*560)

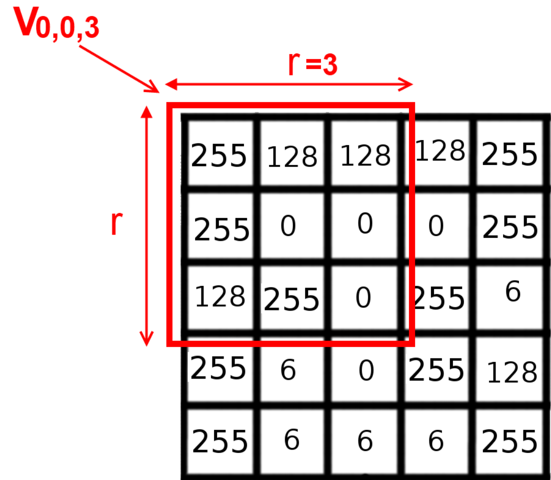


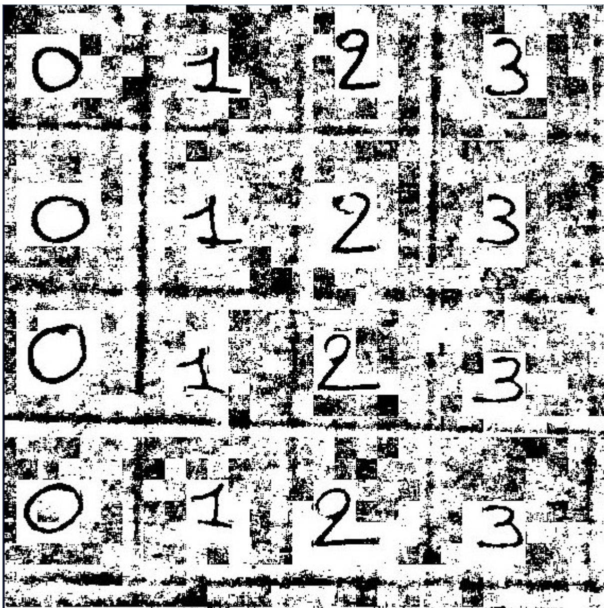


Seuillage global

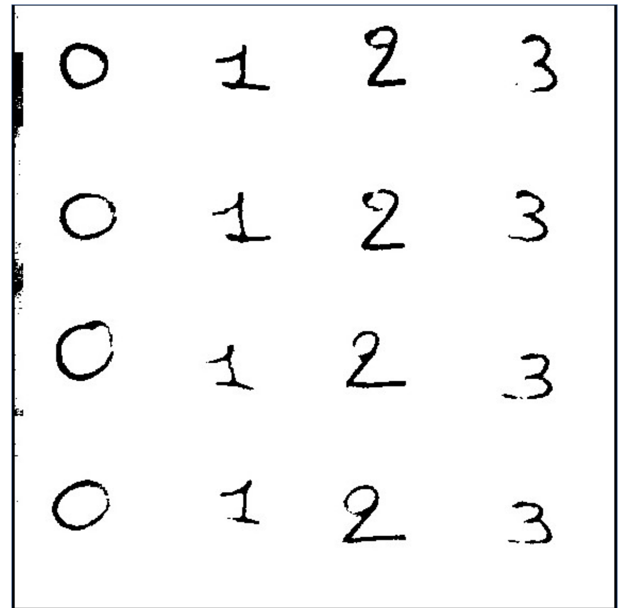
## b) Seuillage relatif :

- $V_{i,j,r}$ un voisinage du pixel $P_{i,j}$
- $S_{i,j,r} = \dfrac{max(V_{i,j,r})+min(V_{i,j,r})}{2}$

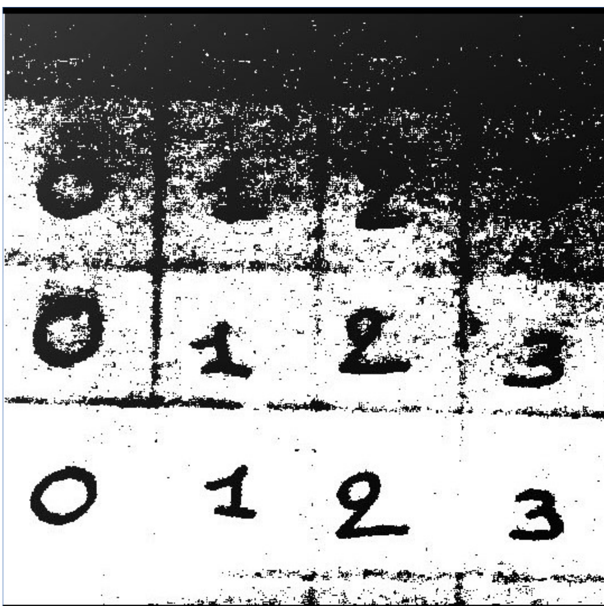


R = 20



R = 180



R = 1800

# I.2 Squelettisation

Algorithme d'Hilditch :

- Érosion des pixels par itérations successives

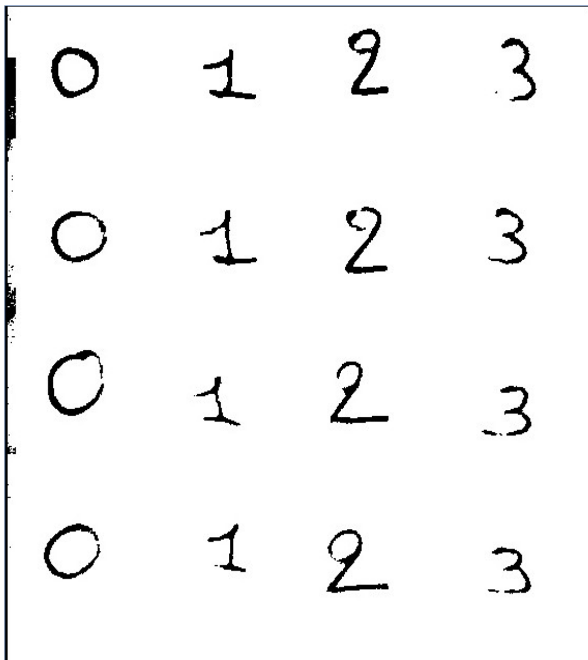


Image source

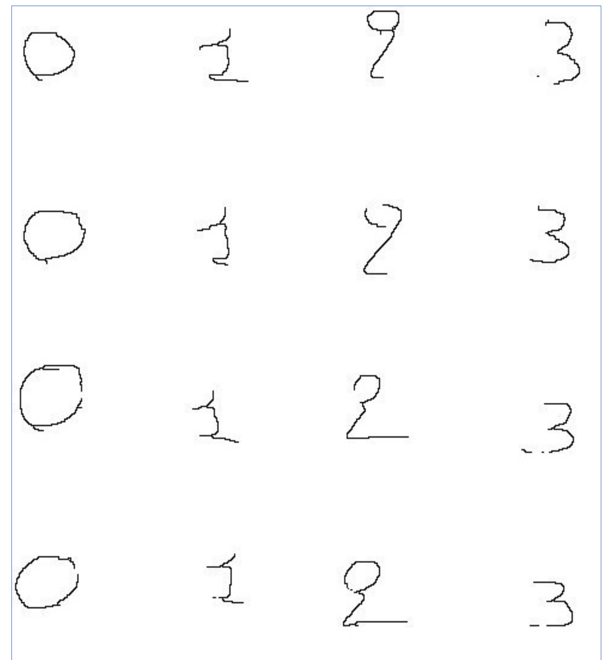
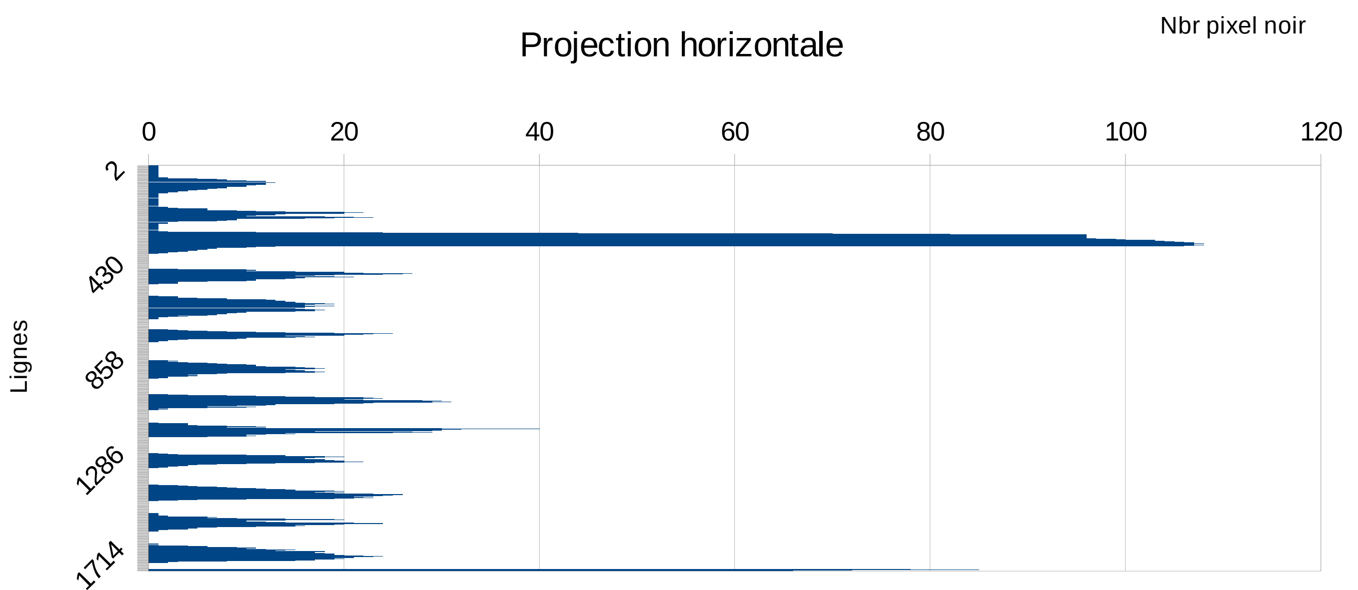


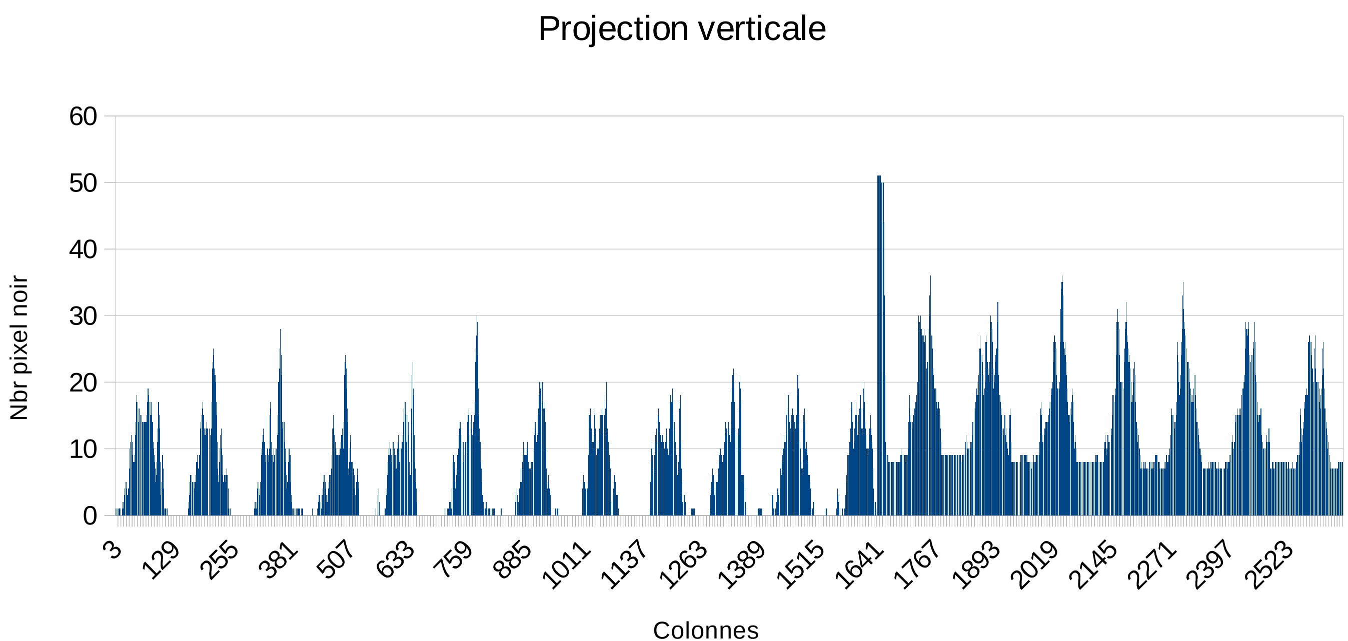
Image squellétisée

# I.3 Segmentation

## Procédé :

- Segmentation en lignes
- Segmentation des lignes en bloc de caractères

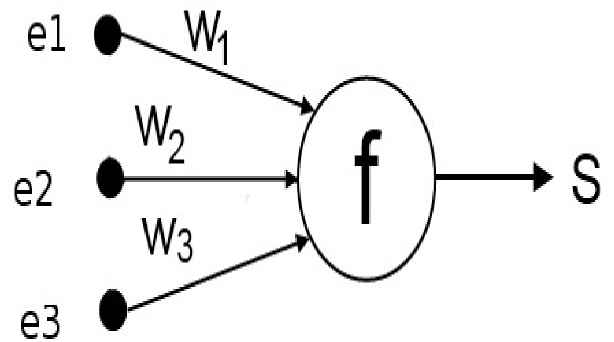


# II.1 Structure

**a) Neurone :**

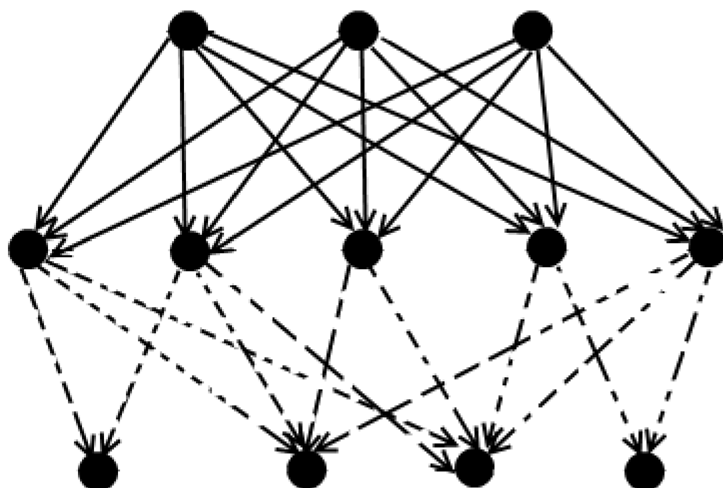
- Un tableau de poids
- Une fonction de transfert
- Une valeur de sortie




Neurone à trois entrées

**b) Multi Layer Perceptron :**

- Une couche d'entrée de 256 neurones
- Une(des) couche(s) cachée(s)
- Une couche de sortie de 10 neurones (1 par classe)



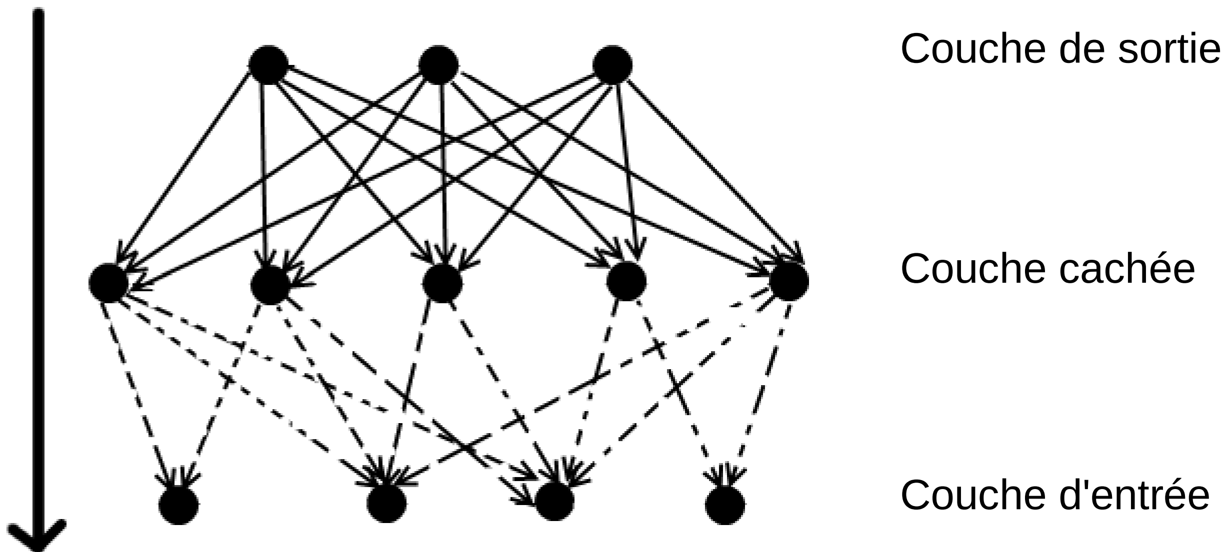
Couche de sortie

Couche cachée

Couche d'entrée

# II.2 Apprentissage

- Calcul d'erreur
- Propagation de celle-ci
- Modification des poids



# II.3 Resultats

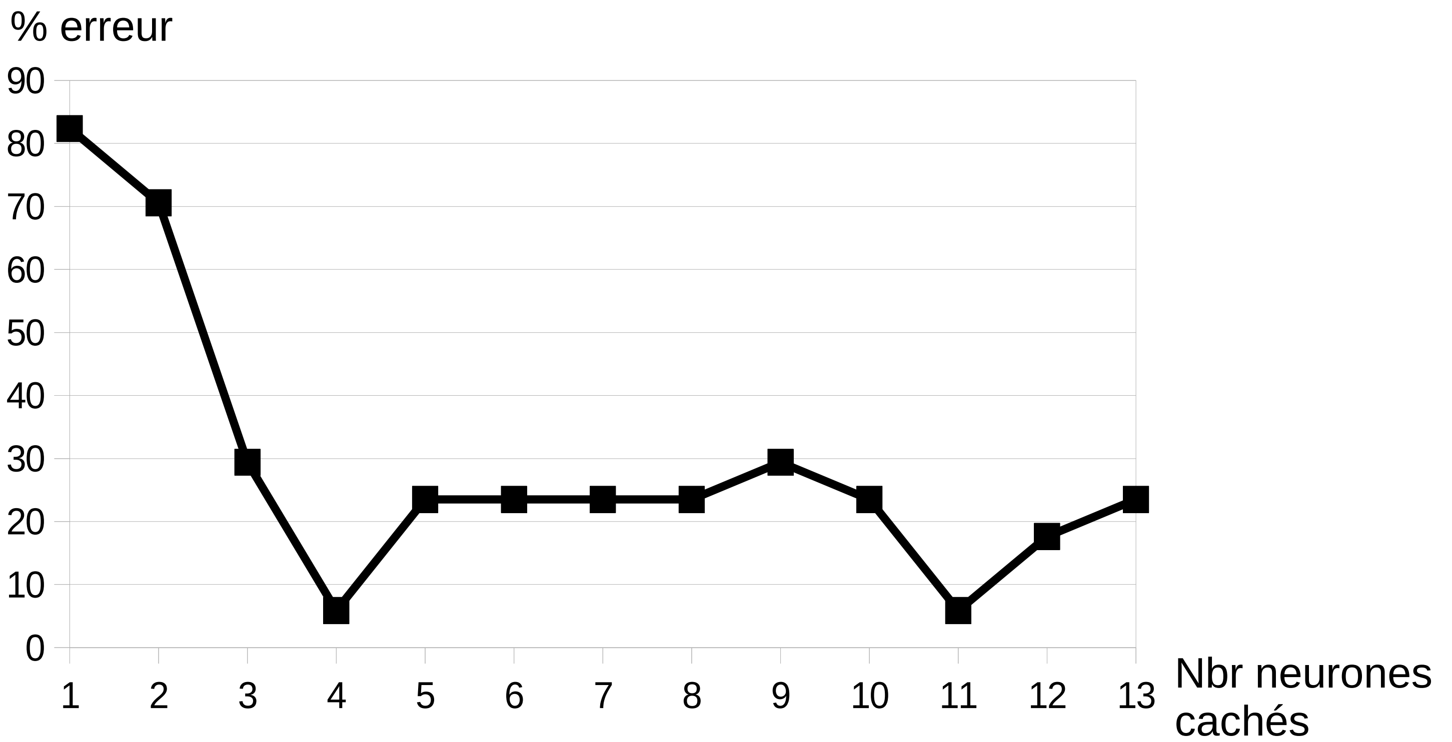
**Base d'apprentissage :**

- 60 Exemples
- 10 Classes

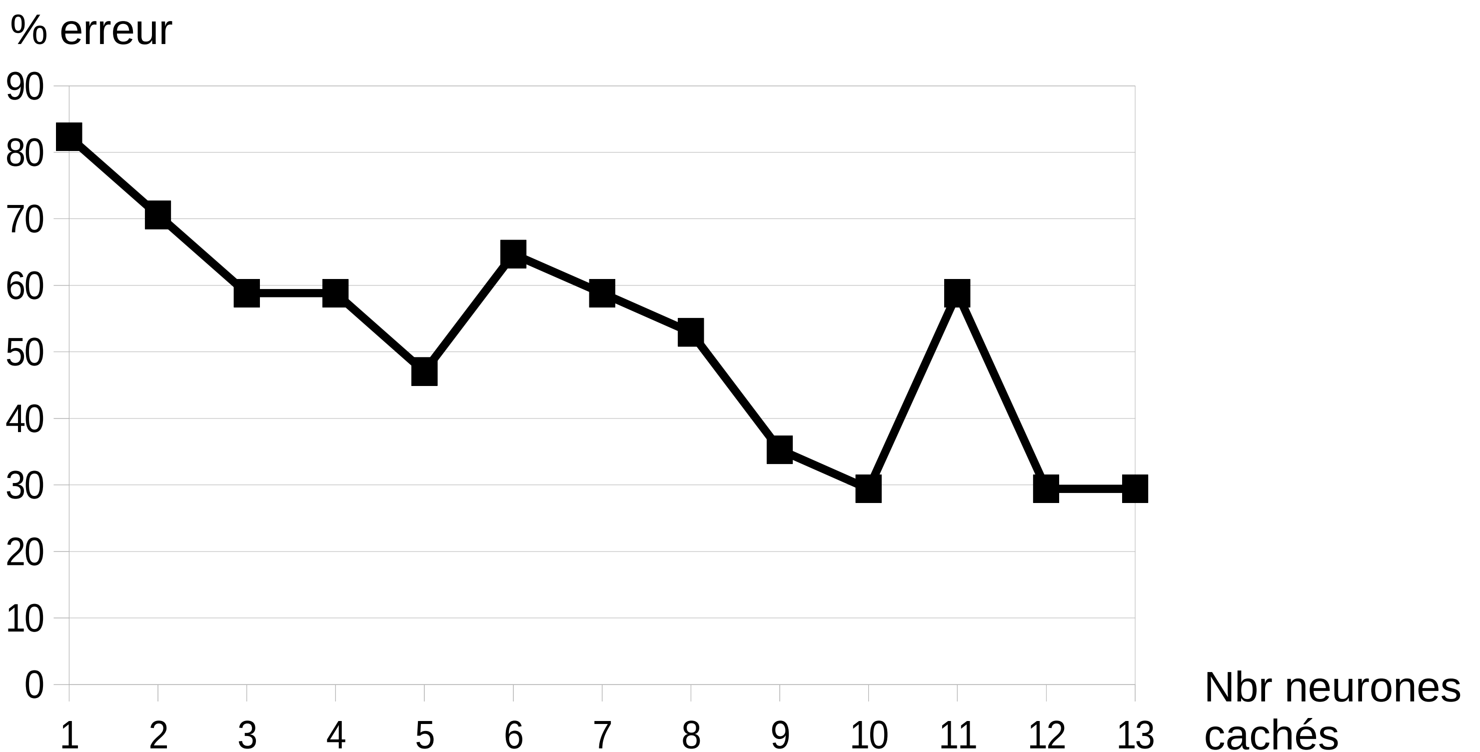
**Méthode de test :**

- Creation d'un reseau ( paramètre :  nbr de neurones)
- Apprentissage
- ➢ Calcul de l'erreur pour chaque exemple

## Erreur pour 10 000 itérations




## Erreur pour 100 itérations



# II.4 Conclusion

**Limitations :**

- Problème de convergence
- Temps d'apprentissage
- Fonctionnement en boîte noire
- Sensibilité aux transformations spatiales

**Avantages** :

- Robustesse
- Peu de pré-traitements nécessaires
- Flexibilité

# Hilditch

## Voisinage d'un pixel P1 :

| P9 | P2 | P3 |
|---|---|---|
| P8 | P1 | P4 |
| P7 | P6 | P5 |

## Conditions de suppression :

- Ni une extrémité, ni un pixel isolé
- Conservation de la connectivité des pixels
- Protection des lignes de 2px de large
- Protection des colonnes de 2px de large

# Rétropropagation

## Erreur (couche de sortie) :

- Valeurs de sortie $(s_i)_{-1<i<n-1}$
- Valeurs attendues $(t_i)_{-1<i<n-1}$
- $e_i = s_i - t_i$

## Rétropropagation :

- Couche courante indice neurone : **i**
- Couche suivante indice neurone : **j** (lower)
- Couche précédante indice neurone : **k** (upper)
- Poids : $w_{i,j}$

$$a_j = \sum_{k=0}^{p-1} w_{k,j} \times s_k$$

$$y_j = f'(a_j) \times (\sum_{i=0}^{m-1} w_{i,j} \times y_i)$$

## Modifications des poids :

$$w_{i,j}(t+1) = w_{i,j}(t) + u \times y_j \times f(a_j)$$

# Neurone

## Fonction d'activation:

- Valeurs d'entrées $\left(e_i\right)_{-1<i<n-1}$
- Les poids $\left(w_i\right)_{-1<i<n-1}$
- $f\left(x\right)=\dfrac{1}{1+\exp\left(o\times\sum_{i=0}^{n-1}\left(e_i\times w_i\right)\right)}$

```
class Neurone{
    protected :
        TransferFunction* fct;
        double output;
        vector<double> weights;
        int nbrWeights;

    public :
        static const int BIAS_VALUE = 0.5;

        void initWeights();
        virtual double calculate(double* inputs);
        virtual void updateWeight(int i, double
const& deltaW);
};
```

# Couche de neurones

```
class NLayer{
    protected :
        int type;
        int size;
        int nbrBias;
        double learningRate;
        double *result;
        double *errors;

        TransferFunction* fct;

        Neurone* *neurones;
        NLayer *upper;
        NLayer *lower;

    public :
        void init(int const*& structure);
        virtual void calculate(double*& inputs);
        virtual void feedback(double*& target);
        virtual void propagate();
        virtual void updateWeights();
};
```

# MLP

```
class NN{
    protected :
        int nbrLayers;
        double learningRate;
        double errorCriterion;
        int criterionType;

        TransferFunction* fct;

        vector< NLayer* > layers;
        NLayer *inputLayer;
        NLayer *outputLayer;

    public :
        void calculate(double* inputs);
        void build(vector< const int* > structure);
        virtual void learn( double* expectedOuputs
);
class MLP : public NN{
    public :
        MLP(vector< const int* > structure, double
errorCriterion, double learningRate=0.5, int
criterionType=1);

        void build(vector< const int* > structure);
        void learn( double* expectedOuputs );
};
```