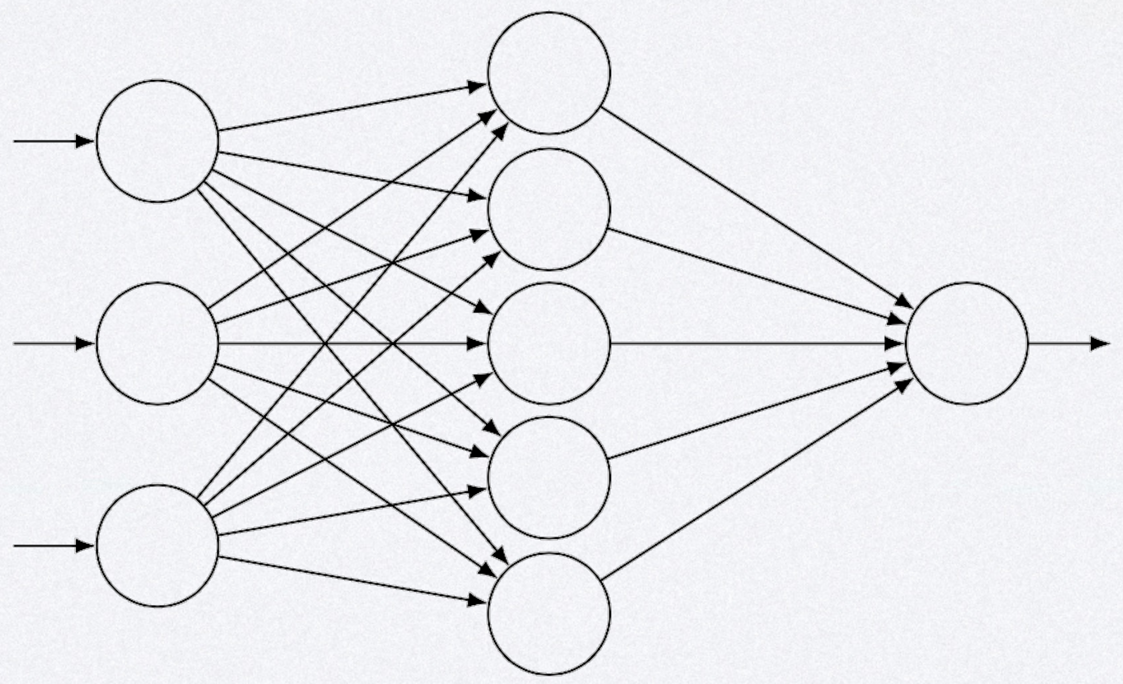
SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

ISUP

RÉSEAUX DE NEURONES

Présentation du rapport - M2 Ingénierie Statistique et Data Science

Raphaël Mignot - Institut de Statistique de l'Université de Paris



Avril 2020

SOMMAIRE

I. PERCEPTRON MULTI-COUCHES

- Problème d'apprentissage supervisé : on souhaite estimer la relation entre une entrée $x \in \mathbb{R}^d$ et une sortie y. On cherche f qui minimise la perte $l(\hat{y}, y)$ où $\hat{y} := f(x)$.

- Perceptron = architecture la plus basique, suite de transformations linéaires et d'application d'une fonction dite d'activation. Un vecteur x entre dans un neurone, on lui applique $a := W^T x + b$ puis on obtient la sortie $h := g(a)$ avec g la fonction d'activation.

- Hyper-paramètres : matrice des poids W, vecteur de biais b. Une valeur par neurone, qui évolue au fil des époques (*epoch*).

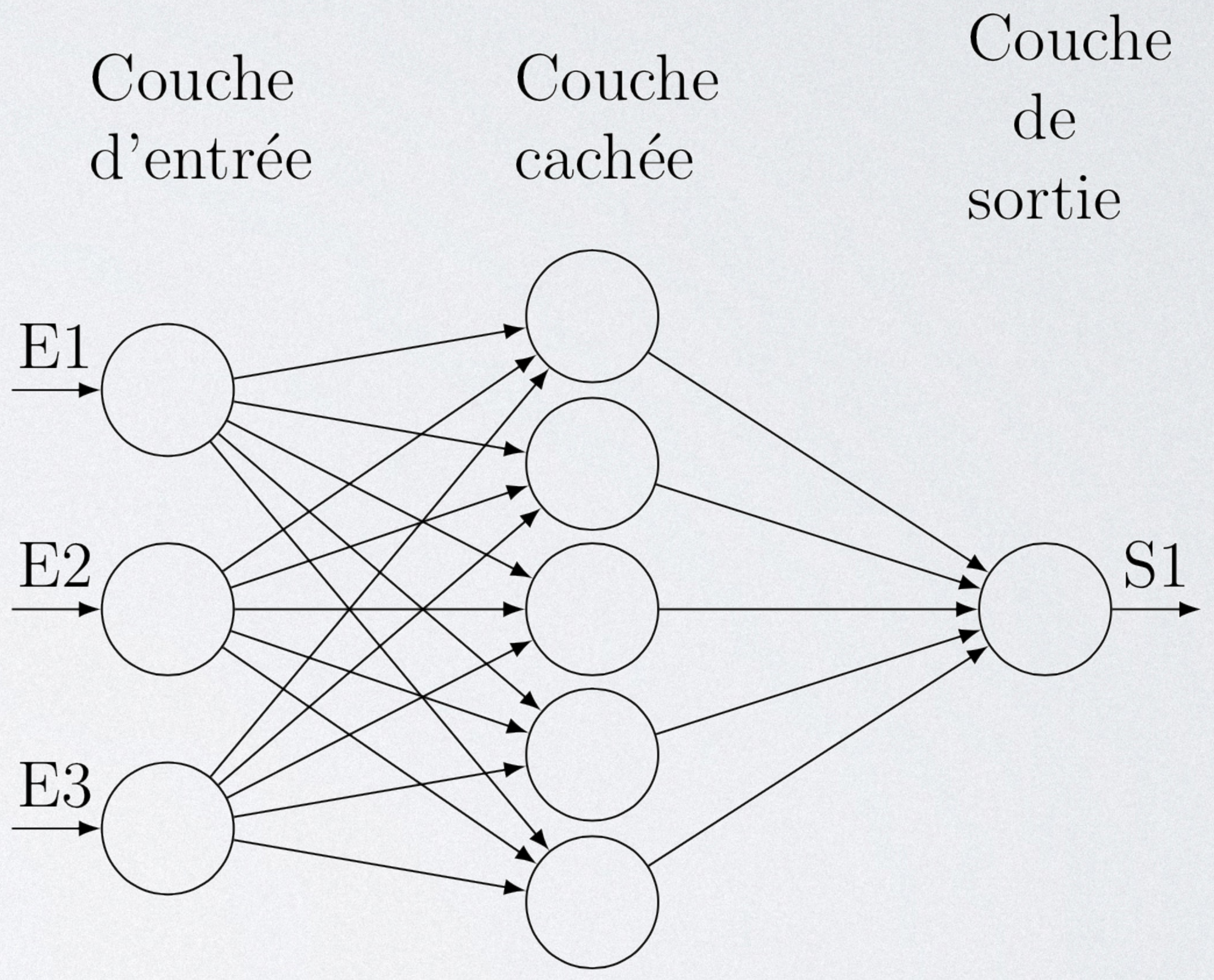


Fig. 1 : Perceptron à une couche cachée

APPRENTISSAGE ET RÉTRO-PROPAGATION

- Propagation de x : obtention de la sortie $\hat{y}$ avec laquelle on calcul le coût $J(x, \theta)$.

- Rétro-propagation : méthode pour calculer le gradient $\nabla_\theta J$ et mettre à jour les paramètres θ. On note $w^k_{i,j}$ le poids entre le neurone j de la couche $k-1$ et le neurone i de la couche k.

- Avantage : peu coûteux en calcul et donc extrêmement rapide.

- Mise à jour des poids et des biais par descente de gradient (par ex.) : $\theta^{t+1} = \theta^t - \alpha \dfrac{\partial J(x, \theta^t)}{\partial \theta}$.

$$\frac{\partial J}{\partial w^k_{i,j}} = \frac{\partial J}{\partial a^k_j}\frac{\partial a^k_j}{\partial w^k_{i,j}} \quad \text{et} \quad \frac{\partial a^k_j}{\partial w^k_{ij}} = \frac{\partial}{\partial w^k_{ij}} \sum_l w^k_{lj} h^{k-1}_l = h^{k-1}_i$$

donc $$\frac{\partial J}{\partial w^k_{i,j}} = h^{k-1}_i \frac{\partial J}{\partial a^k_j}$$

calculé à l'étape précédente

h^{k-1}_i : valeur stockée en mémoire

$$\frac{\partial J}{\partial a^k_j} = \sum_l \frac{\partial J}{\partial a^{k+1}_l}\frac{\partial a^{k+1}_l}{\partial a^k_j}$$

g' connue (simple)

$$\frac{\partial a^{k+1}_l}{\partial a^k_j} = w^{k+1}_{jl} g'(a^k_j)$$

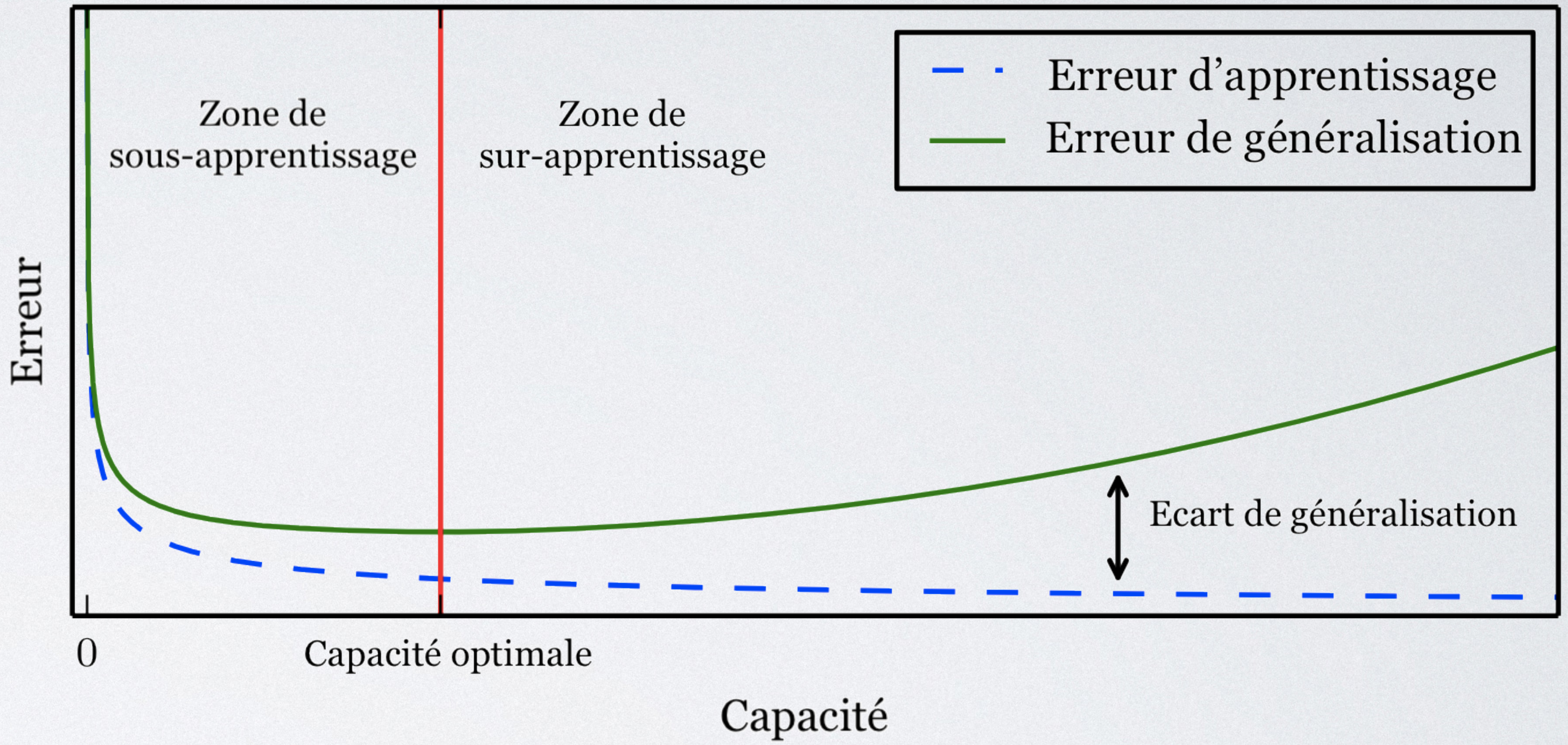
Principe de la rétro-propagation

RÉGULARISATION EN APPRENTISSAGE PROFOND

- Régularisation = toute méthode qui diminue l'erreur de généralisation sans faire baisser l'erreur d'apprentissage i.e. qui réduit le sur-apprentissage (cf figure).

- Méthodes essentielles en apprentissage automatique.

- De multiples techniques : régression d'arête (*ridge regression*), pénalisation L^1, arrêt prématuré (*early stopping*), bagging, dropout, etc.



$$\tilde{J}(\theta, x, y) = J(\theta, x, y) + \alpha\Omega(\theta)$$

avec $\Omega(\theta) = \frac{1}{2}||w||_2^2$

ou $\Omega(\theta) = ||w||_1$

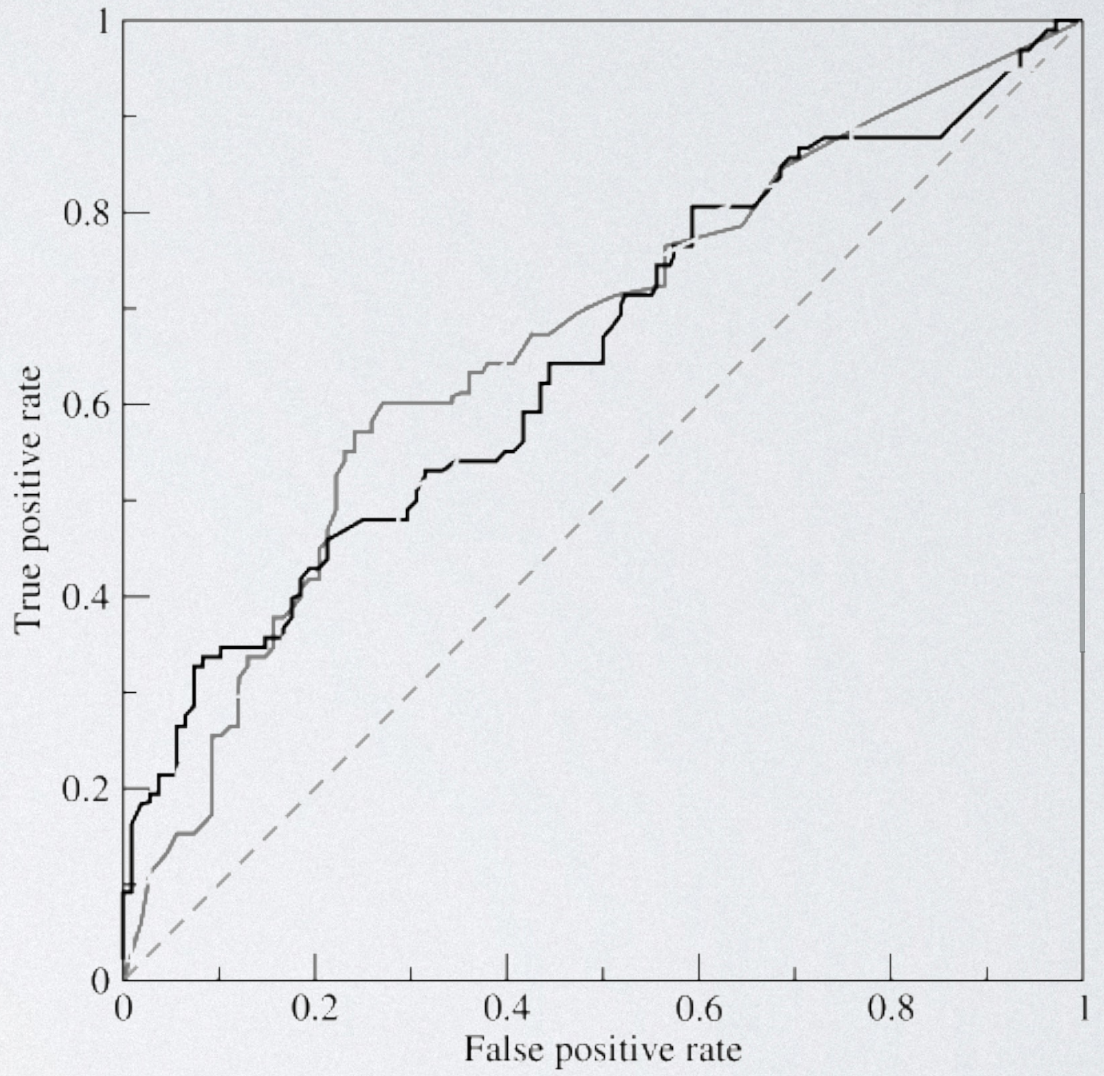
RÉSUMÉ SUR LES CARACTÉRISTIQUES D'UN RÉSEAU DE NEURONES

- Architecture (PMC, réseau convolutionnel, réseau récurrent, fonctions d'activation, etc.).
- Fonction de perte : la fonction à optimiser (moindres carrés, entropie croisée, etc.).
- Méthode d'optimisation : descente de gradient stochastique, ADAM, etc.
- Métrique : pour évaluer les performances et le comparer avec d'autres algorithmes (justesse, ROC AUC, F-score, etc.).

$$\boxed{\mathrm{softmax}(a_i) := \frac{e^{a_i}}{\sum_{j=1}^{K} e^{a_j}}}$$

Exemple de fonction d'activation pour la couche de sortie

Exemple de métrique : courbe ROC



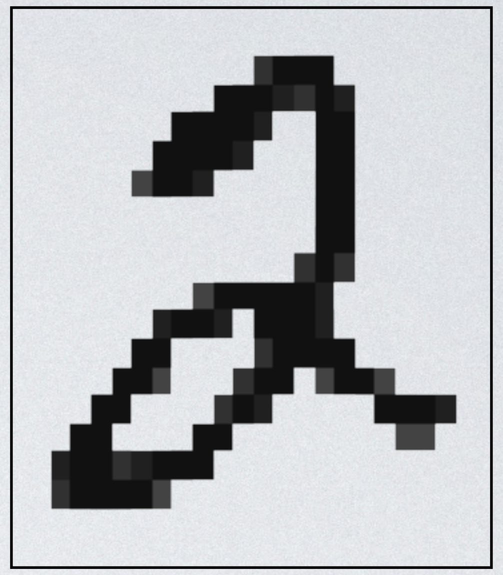
$$\boxed{\text{Justesse} = \frac{\text{Nb de prédictions correctes}}{\text{Nb d'observations}}}$$

Exemple de métrique

APPLICATION PRATIQUE

- On utilisera le jeu de données de chiffres manuscrit MNIST pour toutes les méthodes présentées dans le rapport. Il s'agit de 60000 images de taille 28x28 à classer.

- Les calculs sont effectués avec le logiciel TensorFlow (très rapide, documentation très fournie, régulièrement mis à jour) sous python 3.



TensorFlow

Perceptron multi-couche avec Keras : architecture.

```
model = Sequential()
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Perceptron multi-couche avec Keras : apprentissage.

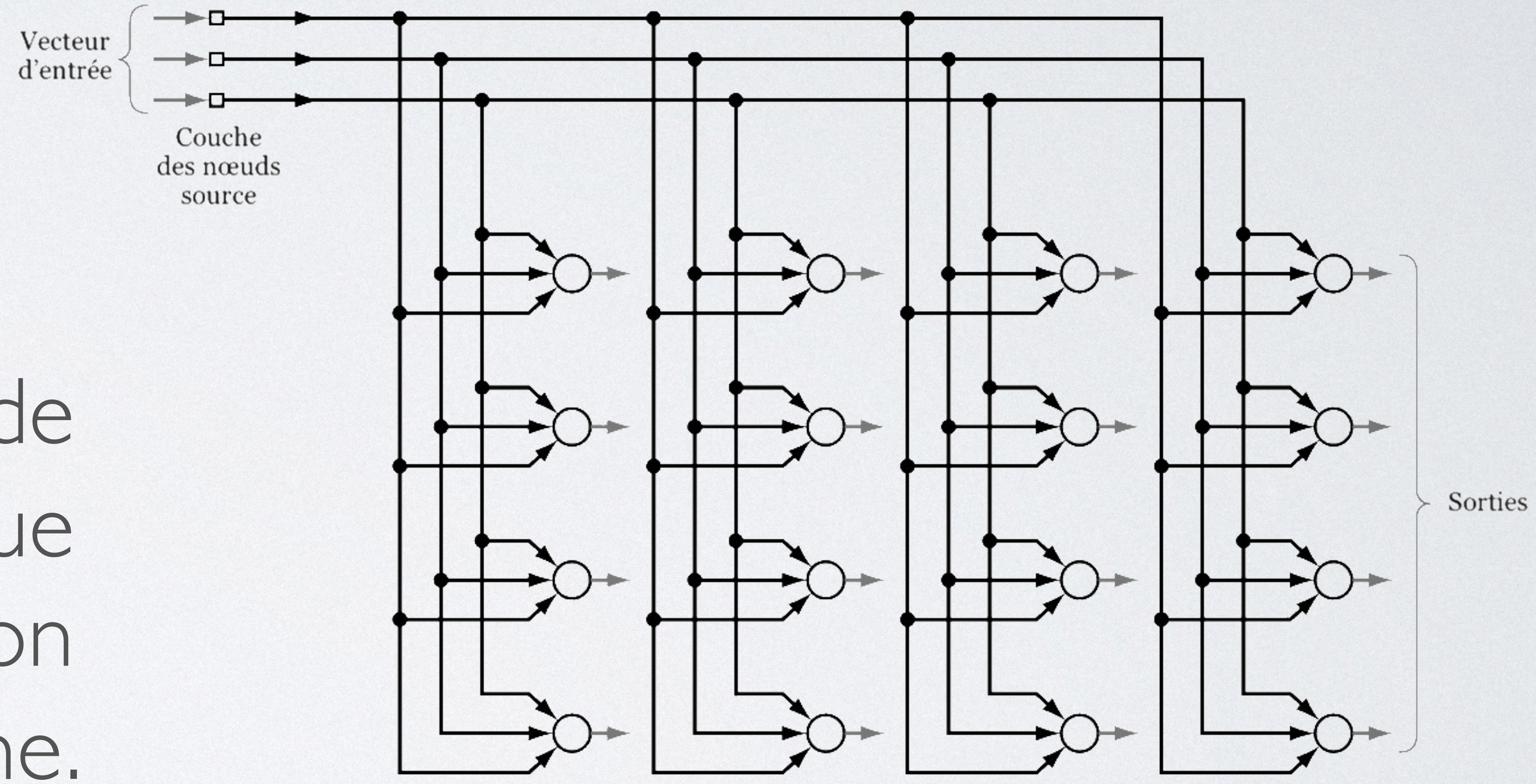
```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))

score = model.evaluate(x_test, y_test, verbose=0)
```

II. CARTE DE KOHONEN

- Apprentissage non supervisé. Utile pour traiter des données de grande dimension. Algorithme simple.

- Permet aussi d'effectuer de la prédiction.

- Des neurones sont disposés sur une grille de taille $N \times N$ (cf schéma ci-contre). Chaque neurone est connecté à toutes les entrées, on note w_i le poids qui connecte x_i au i^e neurone. w_i est de même dimension que les entrées. L'algorithme s'effectue en 6 étapes.

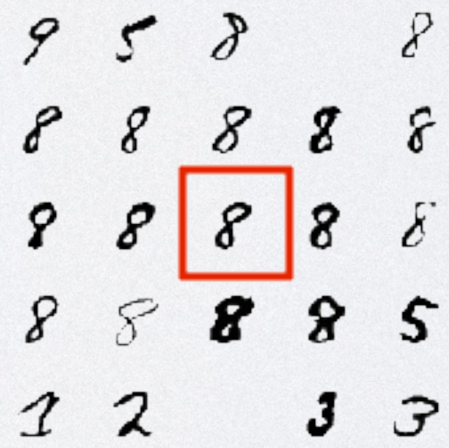


Carte de Kohonen de taille 4×4

ÉTAPES DE L'ALGORITHME

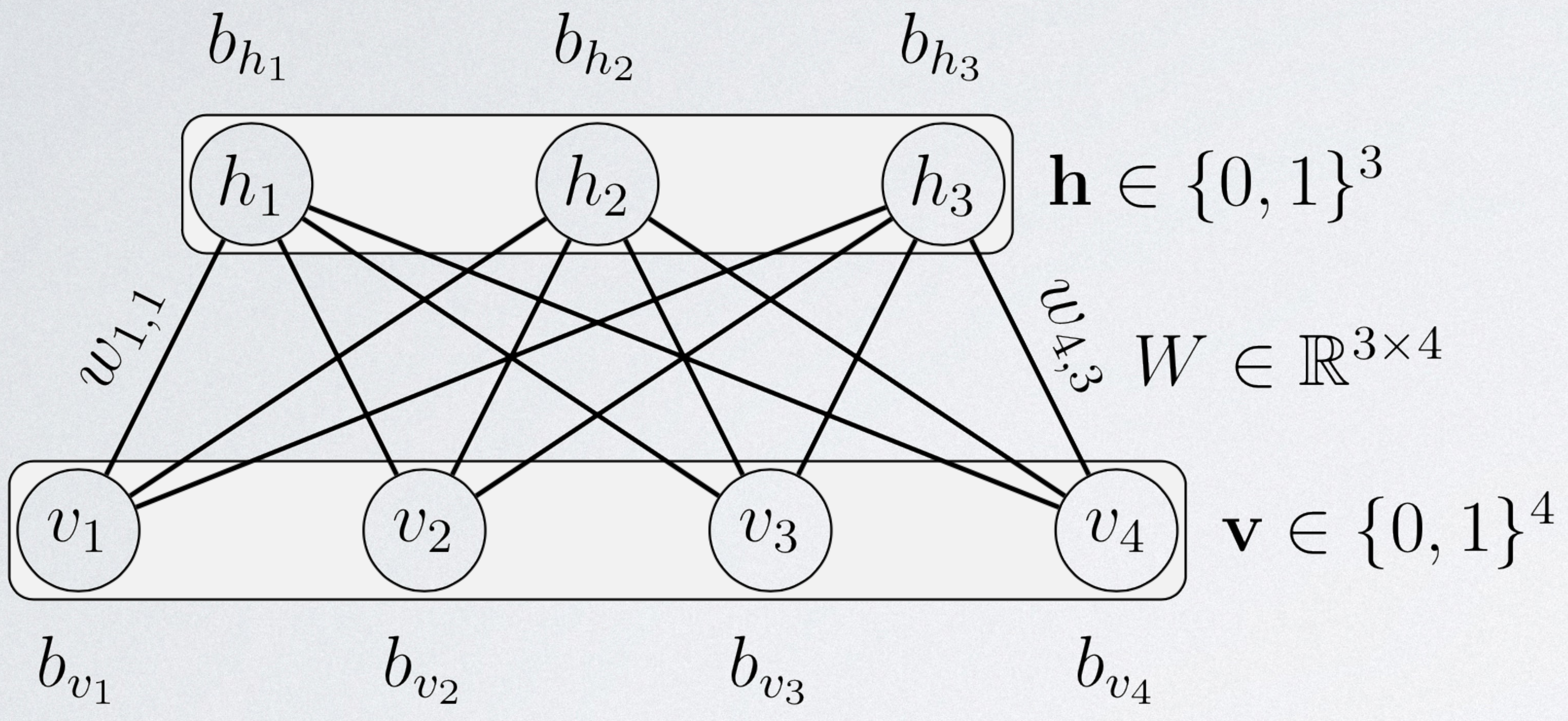
- Initialiser les poids w_i. Initialiser $t = 0$.
- Tirer une observation x aléatoirement dans le jeu.
- Déterminer le neurone I le plus proche : $w_I = \text{argmin}_i ||x - w_i||$ (neurone gagnant ou *best matching unit*).
- Approcher ce neurone I (et ses voisins) de l'observation : $w_i = w_i + \eta(t) h_I(i)(x - w_i), \forall i$ avec η le pas, h la fonction de voisinage (ex. : gaussienne centrée en I).
- $t = t + 1$
- Tant que $t < t_{\text{max}}$, reprendre à l'étape 2.

Carte de Kohonen sur notre jeu MNIST.



Prédiction : détermination du neurone le plus proche pour une nouvelle observation.

III. MACHINE DE BOLTZMANN RESTREINTE



Structure d'une machine de Boltzmann restreinte

$$E(v,h) = -\, b_v^T v - b_h^T h - v^T W h$$

- Apprentissage non supervisé. Réseau de neurones stochastique composé de deux couches : une couche cachée (les h_j) et une couche visible (les v_i).

- Restreinte : on a réduit le nombre de connexions. Un neurone est connecté à tous les neurones de l'autre couche seulement.

- Stochastique : les neurones sont des Bernoulli (donc des v.a.).

- But : estimation de la densité des entrées x_i. Densité de la forme $p(V=v, H=h) = \dfrac{1}{Z}e^{-E(v,h)}$ avec E la fonction énergie, Z la fonction de normalisation.

APPRENTISSAGE

Estimation de l'EMV par l'algorithme de divergence contrastive (CD).

Apprentissage et **divergence contrastive** :

- Phase de propagation : $h^{n+1} = 1$ avec proba $\sigma(v^{(n)T}W + b_h)$
- Phase de reconstruction : $v^{(n+1)} = \sigma(h^{(n+1)T}W + b_v)$

$\Big\} \; CD$

- Application règle apprentissage : $w_{i,j}(t+1) = w_{i,j}(t) + \alpha\left(p(h_i = 1 \mid v^{(0)}) \cdot v_j^{(0)} - p(h_i = 1 \mid v^{(k)}) \cdot v_j^{(k)}\right)$

Les deux premières étapes peuvent être appliquées k fois : on note CD_k. Les poids sont mis à jour seulement après avoir effectué CD_k.

$$
\begin{aligned}
l(W) &= \sum \log P(V = v) \\
&= \sum_v \log \sum_h \exp(-E(v,h)) - \log \sum_{v',h'} \exp(-E(v',h'))
\end{aligned}
$$

On obtient : $\dfrac{\partial \ell(W)}{\partial w_{i,j}} = \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}$

Règle d'apprentissage :

$$
\begin{aligned}
w_{i,j}(t+1) = {} & w_{i,j}(t) \\
& + \alpha(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model})
\end{aligned}
$$

avec α le pas. (c'est une règle locale)

IV. RÉSEAUX CONVOLUTIONNELS

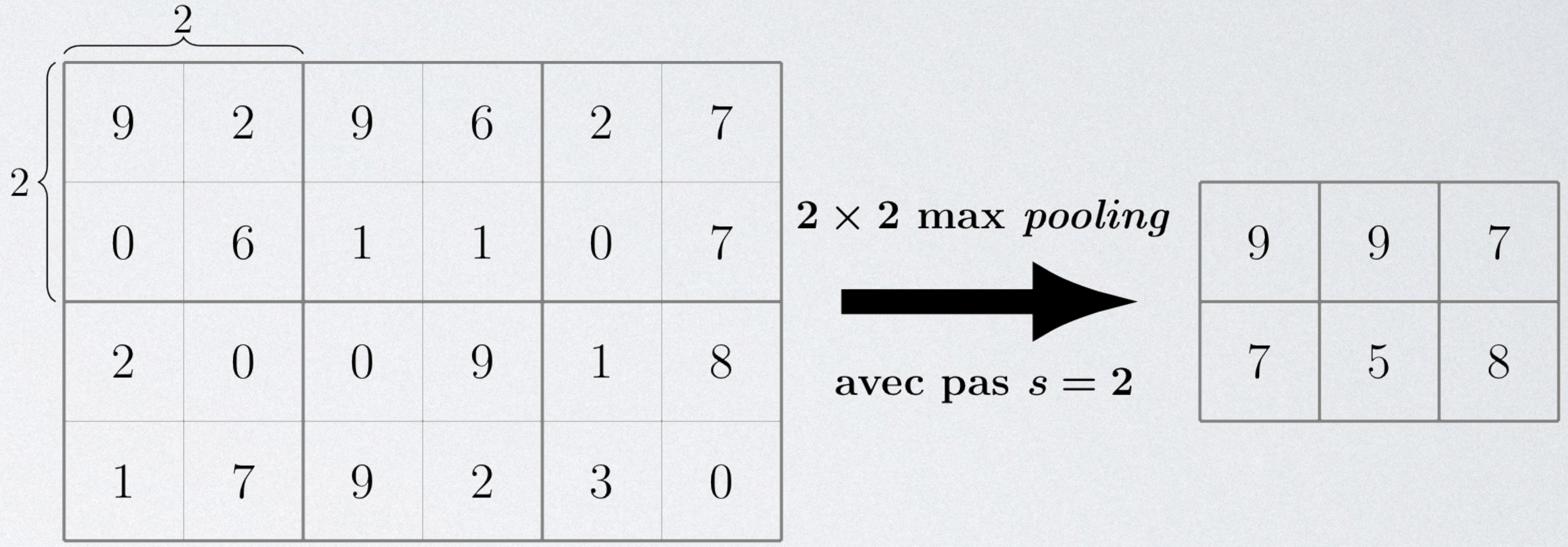
- Réseau qui comprend une **couche de convolution** :
  - Opération de convolution.
  - Détecteur (application fonction non linéaire).
  - Opération de groupage (ou *pooling*).

- Convolution pour une image I de taille $M \times N$:

$$(I * K)(i,j) = \sum_{m=1}^{M} \sum_{n=1}^{N} I(i-m, j-n)K(m,n) \quad \text{avec}$$

  K le noyau (ou filtre).

- *Pooling* : passer un masque de largeur l sur la matrice de pixels avec un pas s (cf. figure ci-contre). Créé une représentation invariante par petites translations de l'entrée.



Pooling avec réduction de dimension (taille 2×2 et pas de 2)

UTILISATION PRATIQUE

Réseau convolutionnel avec Keras : architecture.

TensorFlow

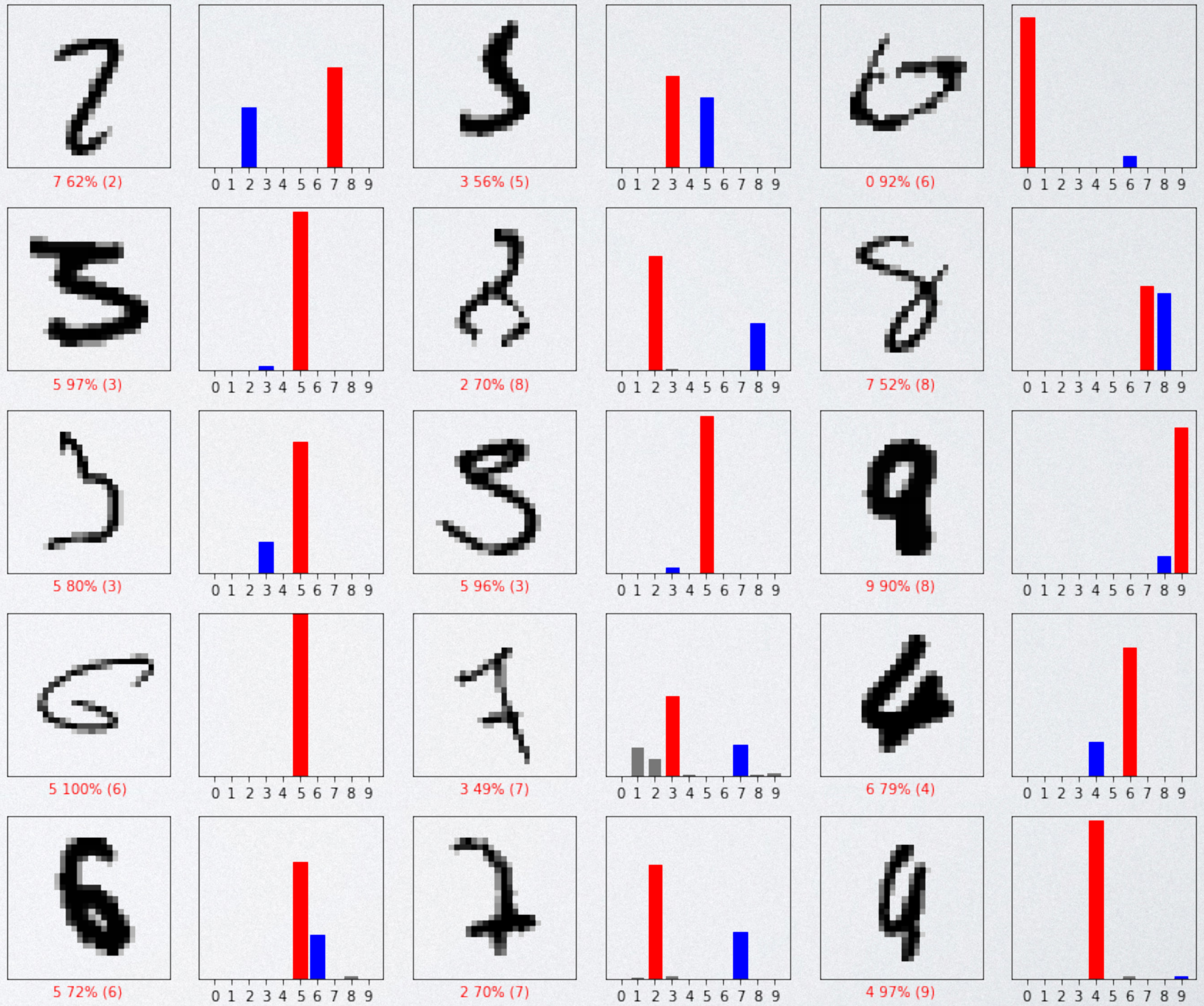
```
model = Sequential()
model.add(Conv2D(filters=32,                    #output size
                 kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape,
                 padding='valid',               #default
                 stride=(1,1)))                 #default
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2),
                       strides=None,            #use pool_size
                       padding='valid'))        #default
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Réseau convolutionnel avec Keras : phase d'apprentissage.

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))

score = model.evaluate(x_test, y_test, verbose=0)
```

```
Epoch 12/12
60000/60000 [==============================] - 6s 101us/step -
loss: 0.0263 - acc: 0.9913 - val_loss: 0.0270 - val_acc: 0.9914
Test loss: 0.02704868172882725
Test accuracy: 0.9914
```

Sortie de notre algorithme (99.1% de bonne classification).



Exemples de mauvaises classifications par le réseau conv.

CONCLUSION