

# Big Data Mining

## Réseaux de Neurones

### Master 2 BIBD et Master 2 SISE

Guillaume Metzler

guillaume.metzler@univ-lyon2.fr



**INSTITUT  
de la  
communication**



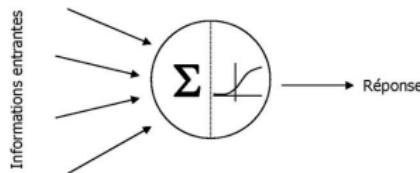
Université de Lyon, Lyon 2, ERIC EA3083, Lyon, France

Automne 2020

# Rappel élémentaires sur les réseaux de neurones

# Neurone

Figure 1 – Neurone artificiel (McCulloch et Pitts, 1943)



Un neurone (McCulloch and Pitts, 1943) est un modèle *simple* qui :

- prend en entré un vecteur  $\mathbf{x} \in \mathbb{R}^d$
- calcule une combinaison linéaire des éléments  $\mathbf{x}^j$
- retourne une réponse suite au passage dans une fonction d'activation

$$y = g \left( b + \sum_{j=1}^d w_j \mathbf{x}^j \right),$$

où  $g$  est la fonction d'activation.

# Fonctions d'activation

On peut utiliser plusieurs fonctions d'activations :

- Rampe :  $g(x) = x$
- ReLU :  $g(x) = \max(0, x)$
- ParReLU :  $g(x) = \alpha x \mathbb{1}_{x < 0} + x \mathbb{1}_{x > 0}$
- SoftReLU :  $g(x) = \ln(1 + e^x)$
- Heaviside :  $g(x) = \mathbb{1}_{x > 0}$
- $g(x) = \tanh(x)$
- ...

L'usage des fonctions d'activation va dépendre de votre contexte (classification ou regression) mais aussi des propriétés attendues : monotonie - dérivabilité - smoothness - ou encore de l'endroit dans le réseau.

# Réseau de neurones

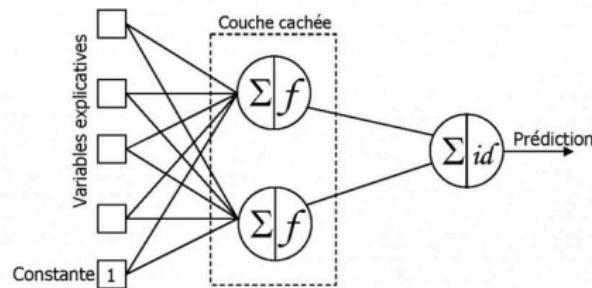
Un réseau de neurones est une succession de neurones qui est caractérisé par :

- le nombre de couches du réseau
- le nombre de neurones à chaque couche
- la dimension de l'output du réseau
- les fonctions d'activations utilisées
- les opérations effectuées (convolution ou somme pondérée)
- la tâche à effectuer (classification, calcul score, régression, ...)

# Réseau multi-couches

Ci-dessous un exemple de réseau dit **fully-connected**, i.e. chaque neurone de la couche suivante est reliée à tous les neurones de la couche précédente. Cela n'est pas une obligation bien sûr ! (voir la notion de **dropout**)

Figure 2 – réseau de neurones



Les couches intermédiaires d'un réseau sont appelés les **couches intermédiaires**.

# Apprentissage d'un réseau

Quelques loss/fonctions d'activations employées pour apprendre un réseau.

- **Regression** : on utilisera l'erreur quadratique avec une fonction d'activation linéaire en sortie

$$\ell(z, \hat{z}(\mathbf{x})) = (z - \hat{z}(\mathbf{x}))^2.$$

- **Classification** : on utilisera la cross entropie avec une fonction d'activation sigmoïde

$$\ell(y, \hat{y}(\mathbf{x})) = -y \log(\hat{y}(\mathbf{x})) - (1 - y) \log(1 - \hat{y}(\mathbf{x})).$$

Dans le cas multi-classe, on aura plusieurs sorties (une par classe) et on utilise une fonction d'activation "soft-max".

# Apprentissage d'un réseau

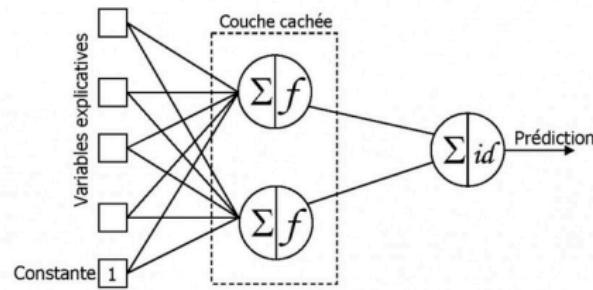
Comme pour tout modèle d'apprentissage machine, l'apprentissage d'un réseau de neurones se fait par rétropropagation du gradient de la couche de sortie vers la couche d'entrée (dérivation de fonctions composées).

Ce calcul peut-être fastidieux à effectuer manuellement mais pour la plupart des software utilisées, il n'est pas nécessaire de déterminer soit même le gradient

## Différents type de réseaux

# Réseaux Profonds ou Feed Forward Network

Figure 2 – réseau de neurones



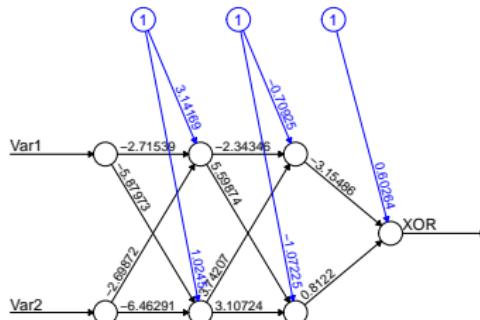
Ces réseaux sont utilisés pour effectuer des tâches de classification ou encore pour faire de la reconnaissance de caractères dans des images ou dans un contexte de speech recognition, on peut aussi l'utiliser pour une tâche de régression

# Réseaux Profonds ou Feed Forward Network

## Pour de la classification

On prends l'exemple ici d'un réseaux avec deux couches cachées de deux neurones. On apprend 5 modèles et on plot le meilleur modèle.

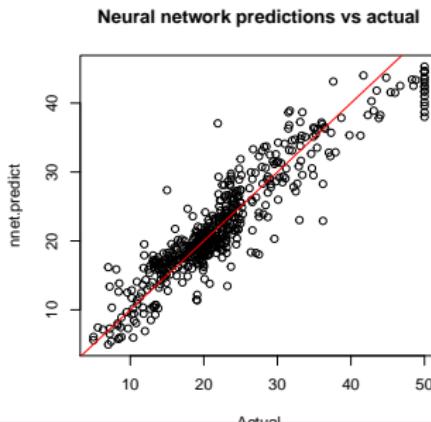
```
1 library(neuralnet)
2
3 XOR <- c(0,1,1,0)
4 xor.data <- data.frame(expand.grid(c(0,1), c(0,1)), XOR)
5 net.xor <- neuralnet( XOR~Var1+Var2, xor.data, hidden=c(2,2), rep=5)
6 plot(net.xor, rep="best")
```



Error: 5.1e-05 Steps: 74

# Réseaux Profonds ou Feed Forward Network

```
1 library(nnet)
2 library(mlbench)
3 data(BostonHousing)
4
5 nnet.fit <- nnet(medv/50 ~ ., data=BostonHousing, size=4, maxit=200)
6 nnet.predict <- predict(nnet.fit)*50
7
8 mean((nnet.predict - BostonHousing$medv)^2)
9
10 plot(BostonHousing$medv, nnet.predict,
11       main="Neural network predictions vs actual",
12       xlab="Actual")
13 abline(a=0,b=1,col="red")
```



# Mnist avec Keras

Un exemple un peu compliqué mais utilisant la librairie *keras* de 

```
1 library(keras)
2 # Quelques options pour faire du dropout
3 FLAGS <- flags(
4   flag_numeric("dropout1", 0.4),
5   flag_numeric("dropout2", 0.3)
6 )
7
8 # Preparation des donnees
9 mnist <- dataset_mnist()
10 x_train <- mnist$train$x
11 y_train <- mnist$train$y
12 x_test <- mnist$test$x
13 y_test <- mnist$test$y
14
15 # On transforme les matrices en vecteurs (peut etre fait a posteriori)
16 dim(x_train) <- c(nrow(x_train), 784)
17 dim(x_test) <- c(nrow(x_test), 784)
18
19 # On rescale maintenant entre [0,1] les valeurs RGB
20 x_train <- x_train/255
21 x_test <- x_test/255
22
23 # On transforme egalement le classes en un vecteur binaire
24 y_train <- to_categorical(y_train, 10)
25 y_test <- to_categorical(y_test, 10)
```

# Mnist avec Keras

On peut s'amuser à regarder les données avant transformation en vecteurs

```
1 par(mfrow=c(4,5))
2 par(mar=c(0,0,2,0),xaxs="i", yaxs="i")
3 for (i in 1:20){
4   img <- x_train[i,,]
5   img <- t(apply(img,2,rev))
6   image(1:28,1:28, img, col=gray((0:255)/255), xaxt="n", yaxt="n")
7 }
```



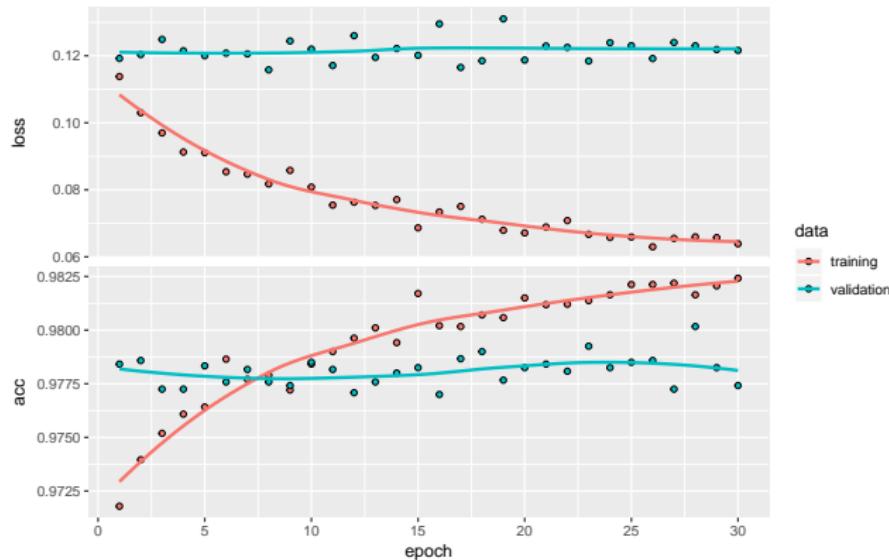
# Mnist avec Keras

## Préparation du modèle

```
1 # Preparation du modèle (structure puis loss et solveur)
2 model <- keras_model_sequential()
3 model %>%
4   # layer_flatten(input_shape = c(28,28)) %>%
5   layer_dense(units = 128, activation = 'relu', input_shape = c(784)) %>%
6   layer_dropout(rate = FLAGS$dropout1) %>%
7   layer_dense(units = 128, activation = 'relu', input_shape = c(784)) %>%
8   layer_dropout(rate = FLAGS$dropout2) %>%
9   layer_dense(units = 10, activation = "softmax")
10
11 model %>% compile(
12   loss = "categorical_crossentropy",
13   optimizer = optimizer_rmsprop (lr = 0.003),
14   metrics = c("accuracy")
15 )
16
17 # Entrainement
18 history <- model %>% fit(
19   x_train, y_train,
20   batch_size = 128,
21   epochs = 30,
22   verbose = 1,
23   validation.split = 0.2
24 )
```

# Mnist avec Keras

Keras nous permet d'observer ce que donne l'entraînement du modèle au fur et à mesure des epoch.



# Mnist avec Keras

## Quelques remarques pour conclure

```
1 score <- model %>% keras::evaluate(  
2   x_test, y_test,  
3   verbose=0  
)  
4  
5 cat('Test loss:', score$loss, '\n')  
6 # Test loss: 0.120972  
7 cat('Test accuracy:', score$acc, '\n')  
8 # Test accuracy: 0.978
```

- un outil puissant permettant de faire des calculs sur plusieurs CPU / GPU
- utilise tensorflow et donc de faire du calcul parallèle  
<https://tensorflow.rstudio.com>

A titre d'exercice, vous pouvez essayer de faire les mêmes expériences en sur le jeu de données Fashion-MNIST.

# Utiliser h2o

Vous pouvez également entraîner des réseaux de neurones (mais pas que !) à l'aide du package **h2o** de .

Vous trouverez sur le lien ci-dessous une documentation  et Python sur l'utilisation de cette librairie avec des exemples simples à mettre en œuvre  
<http://docs.h2o.ai/h2o/latest-stable/h2o-docs/booklets/DeepLearningBooklet.pdf>

- définition d'un modèle et apprentissage
- cross-validation
- grid-search
- ...

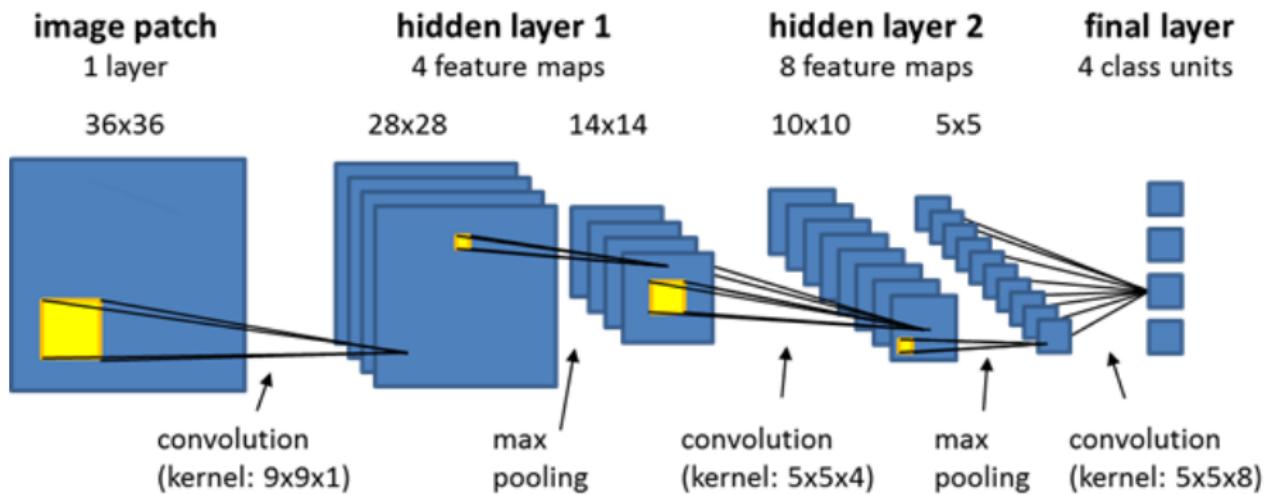
# Limite d'un perceptron multi-couches

Bien que ce type de réseau puisse se révéler très efficace pour certaines tâches, comme on a pu le voir sur **MNIST**, il devient très rapidement limité sur des données dont la dimension est très importante : des images avec une très grande résolution.

La vectorisation de l'image va rapidement entraîner un calcul très important (mémoire et temps) de la part du réseau en phase d'apprentissage (en plus de faire perdre la structure de la donnée).

Pour lutter contre cet aspect là, il est parfois nécessaire d'obtenir, d'apprendre une représentation plus *sparse* de nos données, *i.e.* obtenir un objet "plus petit" mais qui soit suffisamment riche pour effectuer la tâche de classification.

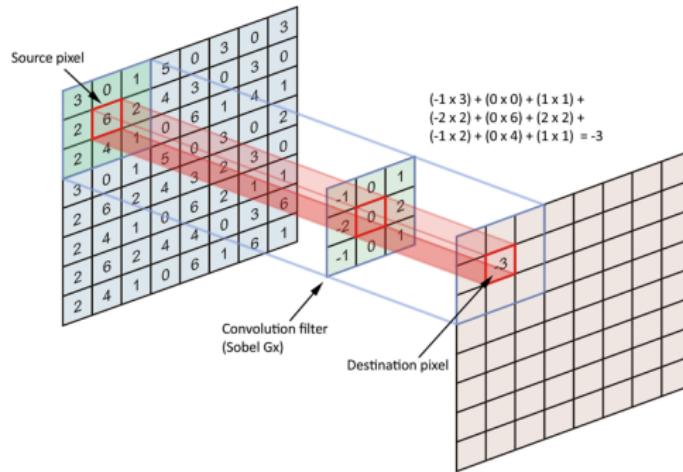
# Réseaux convolutifs



# Réseaux convolutifs

**Des convolutions (filtres)** : pour apprendre à reconnaître des caractéristiques dans l'image + réduction de dimension

**Du pooling** : pour condenser l'information et réduire la dimension des données (max - mean - sum)



# Réseaux convolutifs

## Des applications très variées :

- Détection d'anomalies (Bascol et al., 2017)
- Reconnaissance de motifs (images et vidéos)
- Segmentation sémantique (Fourure et al., 2017) (voir image ci-dessous)
- Analyse d'images médicales



## CNN : de multiples architectures

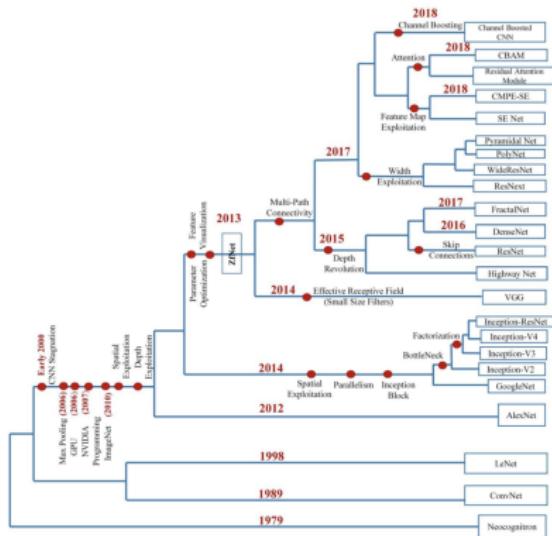


Image extraite de (Khan et al., 2020)

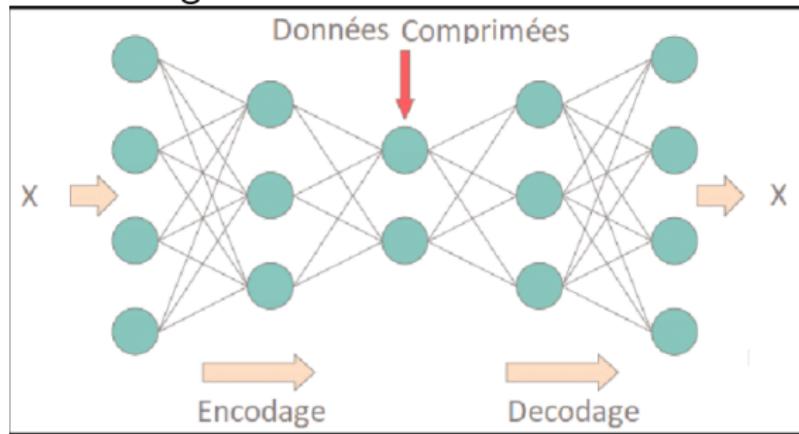
# Réseaux convolutifs

## En pratique :

- Ce genre de réseaux présente une grande complexité, un nombre important de couches (layers) et donc un très grand nombres de paramètres à apprendre.
- Ces modèles présentant un très grand nombre de données nécessitent donc un nombre important de données pendant la phase d'apprentissage ! Des images qui peuvent être de taille conséquente.
- Il est donc parfois intéressant de partir de réseaux connus et pré-entraînés sur des datasets proche de la du dataset utilisé (comme *ImageNet*) et de simplement apprendre les poids du réseaux qui concernent la tâche à effectuer (e.g. de la classification).

# Auto-encodeurs

Un auto-encodeur se décompose de deux parties : une partie **encodage**  $\phi$  et une partie **décodage**  $\psi$ . L'objectif de ce type de réseau (non supervisé !) est d'apprendre une représentation dans un espace de dimension plus petites de vos données. Cette représentation doit cependant permettre de conserver l'information présente dans vos données, i.e. permettre de retrouver les features originelles.



# Auto-encodeurs

Etant donné l'objectif principal des auto-encodeurs, une loss que l'on utilise naturellement est l'erreur quadratique :

$$\ell(\mathbf{x}, \psi \circ \phi(\mathbf{x})) = \|\mathbf{x} - \psi \circ \phi(\mathbf{x})\|^2.$$

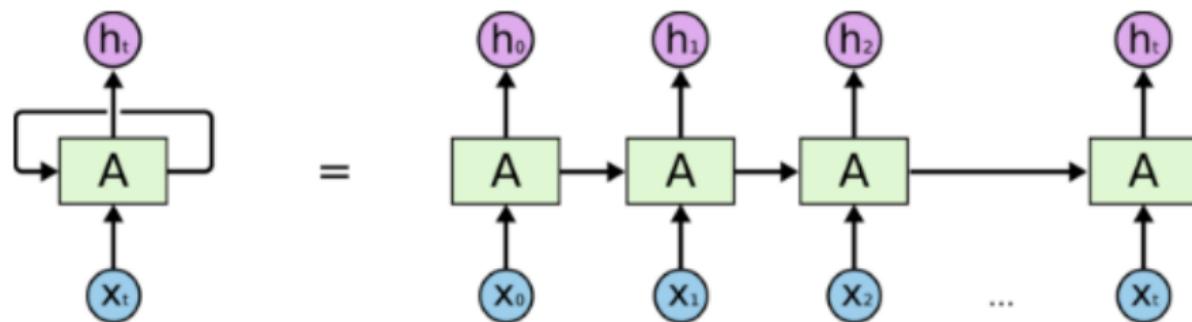
## Applications

- Réduction de dimension
- Détection d'anomalies
- Débruitage de données

**Variantes** : auto-encodeurs variationnels pour l'apprentissage de la distribution des variables latentes (usage de la KL-divergence) utilisés pour des applications médicales en générant des données.

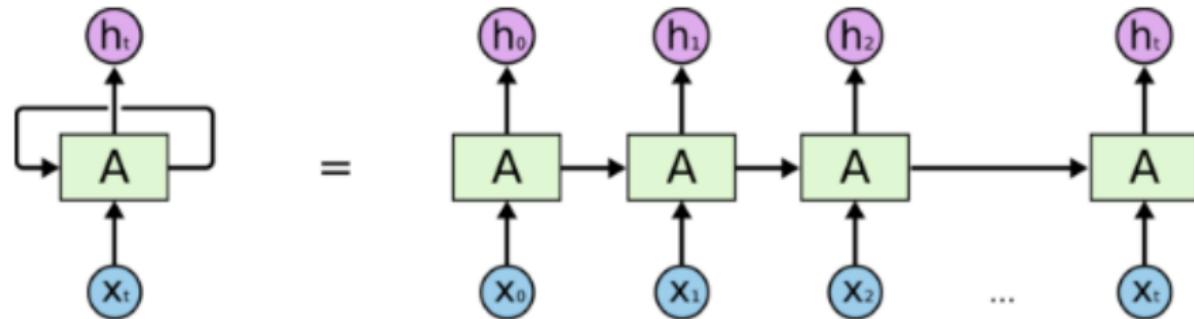
# Réseaux récurrents

Un réseau de neurones récurrents (Rumelhart et al., 1986) est un réseau avec une structure "cyclique". Il peut se représenter comme un même schéma de réseau réseauté juxtaposé les uns à la suite des autres.



An unrolled recurrent neural network.

# Réseaux récurrents



An unrolled recurrent neural network.

Leur structure sous forme de graphe cyclique orienté les rends particulièrement intéressant pour l'étude ou l'analyse de séries temporelles, l'étude de texte (du langage) → **étude de données séquentielles**. En effet ces réseaux ont la particularité de prendre en compte la sortie du "modèle précédent" dans l'estimation de la sortie courante.

# Réseaux récurrents

**Inconvénients :** l'usage d'une structure simple est cependant à proscrire car ce type de réseau présente rapidement un problème de transmission de la *mémoire*.

Plus précisément, plus la séquence d'un texte à étudier est longue, plus l'information se trouvant au début de la séquence aura du mal à se retrouver lors de l'étude de la fin de la séquence.

Ce problème est du à un mauvais "transport" du gradient qui aura tendance à tendre vers 0 plus la séquence à analyser est longue (problème de remontée du gradient le long du réseau).

→ **plusieurs structures existent pour palier à ce problème d'annulation du gradient**

# LSTM : Long Short Term Memory

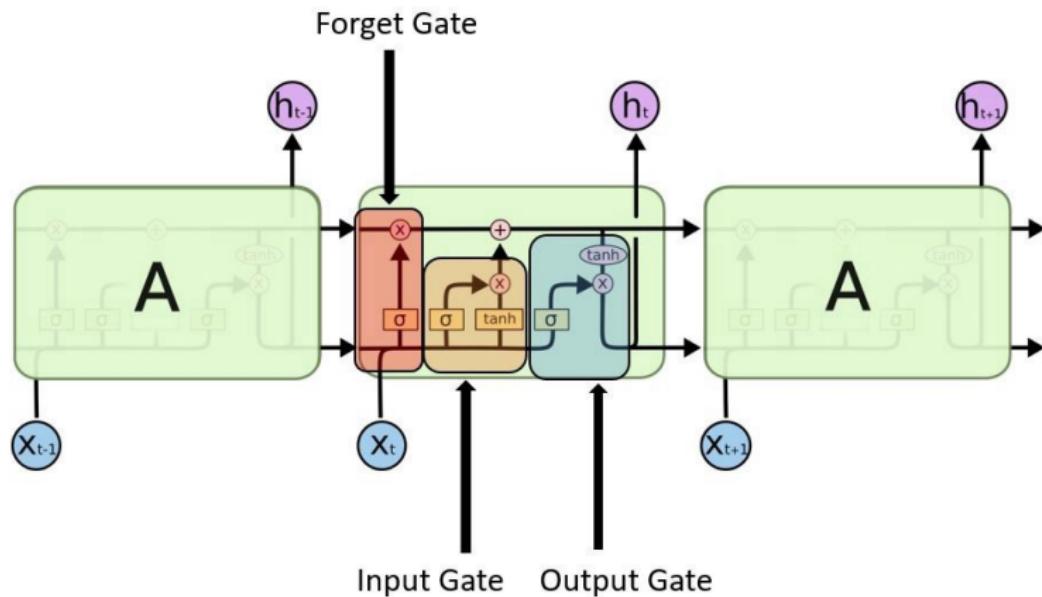


image extraite de

<https://towardsdatascience.com/understanding-rnn-and-lstm-f7cdf6dfc14e>

# LSTM: Long Short Term Memory

## Trois composantes :

- Input Gate : quelles informations et quel poids données à l'information courante que l'on souhaite garder en mémoire
- Forget Gate : on regarde ce que l'on conserve en mémoire de l'étape précédente par rapport à la données qui arrive
- Output Gate : calcul de la sortie en courante en fonction de l'état de la mémoire et la donnée courante

Quelques exemples : Systèmes de recommandations avec des *Time LSTM* (Zhu et al., 2017) ou encore de la classification de signaux avec des *Phased LSTM* (Neil et al., 2016).

# GRU: Gated Recurrent Unit

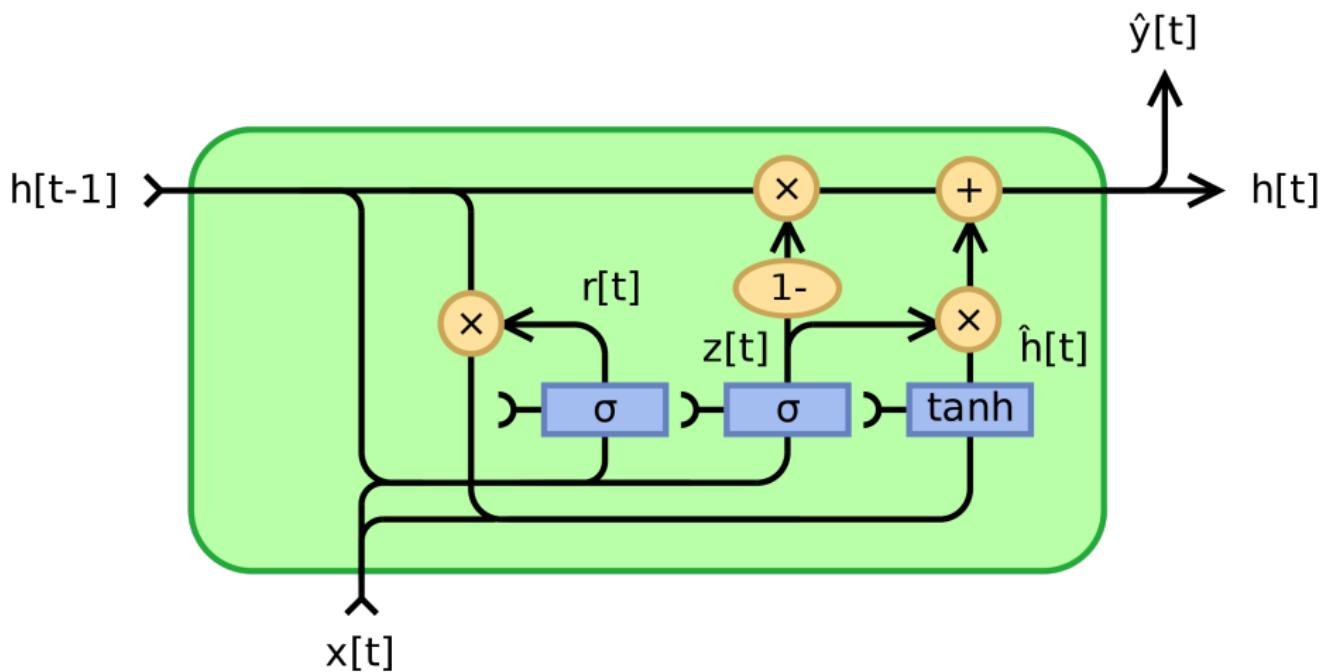
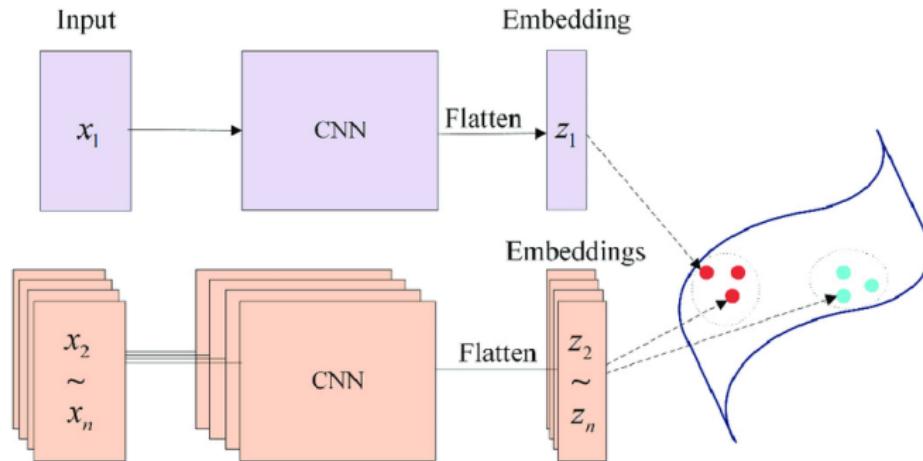


Image extraite de Wikipédia

# Réseaux Siamois

Structure de réseau développée dans les années 90 (Bromley et al., 1994; Chopra et al., 2005). Il se compose de deux réseaux qui partagent **les mêmes poids** et d'une fonction de "similarité" qui va permettre de déterminer si deux exemples sont proches ou non.



# Réseaux Siamois

## Loss

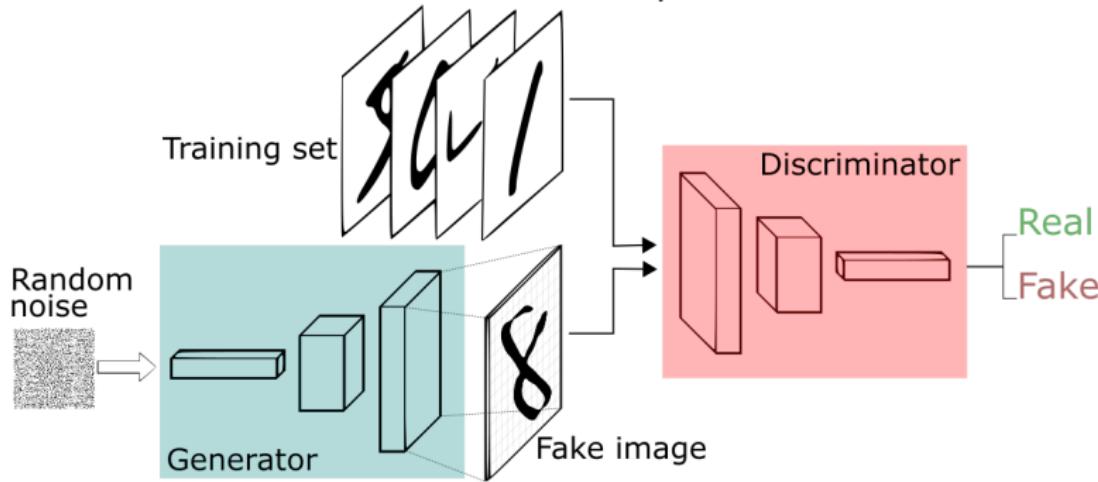
- $\ell(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{2}y_{ij}d(\mathbf{x}_i, \mathbf{x}_j)^2 + \frac{1}{2}(1 - y_{ij})\max(0, m - d(\mathbf{x}_i, \mathbf{x}_j)^2)$
- $\ell d_S(\mathbf{x}, \mathbf{x}_S), d(\mathbf{x}, \mathbf{x}_D) = \max(0, d_S^2 - d_D^2 + m)$

## Applications

- Apprentissage de représentations
- Zero or one (or few) shot learning
- Réduction de dimension
- Détection d'anomalies

# GAN : Generative Adversarial Network

Il s'agit d'un modèle un peu particulier qui met en jeu deux réseaux qui vont se confronter (Goodfellow et al., 2014). Un premier réseau  $G$  va générer des données et un deuxième réseau  $D$  aura pour rôle de *discriminer*.



# GAN : Generative Adversarial Network

La fonction permettant d'entraîner ce type de modèle se présente sous la forme d'un problème min – max

$$\min_G \max_D \ell(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{data}} [\log(D(\mathbf{x}))] + \mathbb{E}_{\mathbf{z} \sim p_{gen}} [\log(1 - D(G(\mathbf{z})))].$$

## Quelques applications

- Faire de la génération de données : texte - audio - photos
- <https://www.thispersondoesnotexist.com> (deepfakes)
- Il existe aussi des applications dans le domaine de la santé ou climat, mais elles sont mineures
- Détection d'anomalies (Bashar and Nayak, 2020)

# References I

- Bascol, K., Emonet, R., Fromont, E., and Debusschere, R. (2017). Improving chairlift security with deep learning. In *International Symposium on Intelligent Data Analysis*, pages 1–13. Springer.
- Bashar, M. A. and Nayak, R. (2020). Tanogan: Time series anomaly detection with generative adversarial networks.
- Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., and Shah, R. (1994). Signature verification using a "siamese" time delay neural network. In *Advances in neural information processing systems*, pages 737–744.
- Chopra, S., Hadsell, R., and LeCun, Y. (2005). Learning a similarity metric discriminatively, with application to face verification. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 539–546. IEEE.
- Fourure, D., Emonet, R., Fromont, E., Muselet, D., Tréneau, A., and Wolf, C. (2017). Residual conv-deconv grid network for semantic segmentation. In *Proceedings of the British Machine Vision Conference, 2017*.
- Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680.
- Khan, A., Sohail, A., Zahoor, U., and Qureshi, A. S. (2020). A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*.

## References II

- McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.
- Neil, D., Pfeiffer, M., and Liu, S.-C. (2016). Phased Istm: Accelerating recurrent network training for long or event-based sequences. In *Advances in neural information processing systems*, pages 3882–3890.
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533–536.
- Zhu, Y., Li, H., Liao, Y., Wang, B., Guan, Z., Liu, H., and Cai, D. (2017). What to do next: Modeling user behaviors by time-lstm. In *IJCAI*, volume 17, pages 3602–3608.