M1DE : Outils et Méthodes de l'IA

- Les réseaux de neurones
- Réseaux de neurones : MLP
- Réseaux de convolution : CNN
- Long Short-Term Memory : LSTM
- Encodeur-Décodeur

Les réseaux de neurones

- Neurone : Perceptron
- La descente de gradient

Neurone : Perceptron

Notions de base

- *Le cerveau est constitué de cellules (neurones) interconnectées.*
- *Les neurones reçoivent des impulsions électriques par l'intermédiaire des dendrites et retournent une information le long d'axones.*
- *Les liens entre axones et dendrites sont gérés par les synapses de l'ordre de plusieurs milliers par neurones.*
- *Ces liens ne peuvent être codés (ADN), ils sont liés à l'apprentissage.*
- *Les signaux traités par les neurones ne sont pas linéaires mais dépendent d'un effet de seuil.*

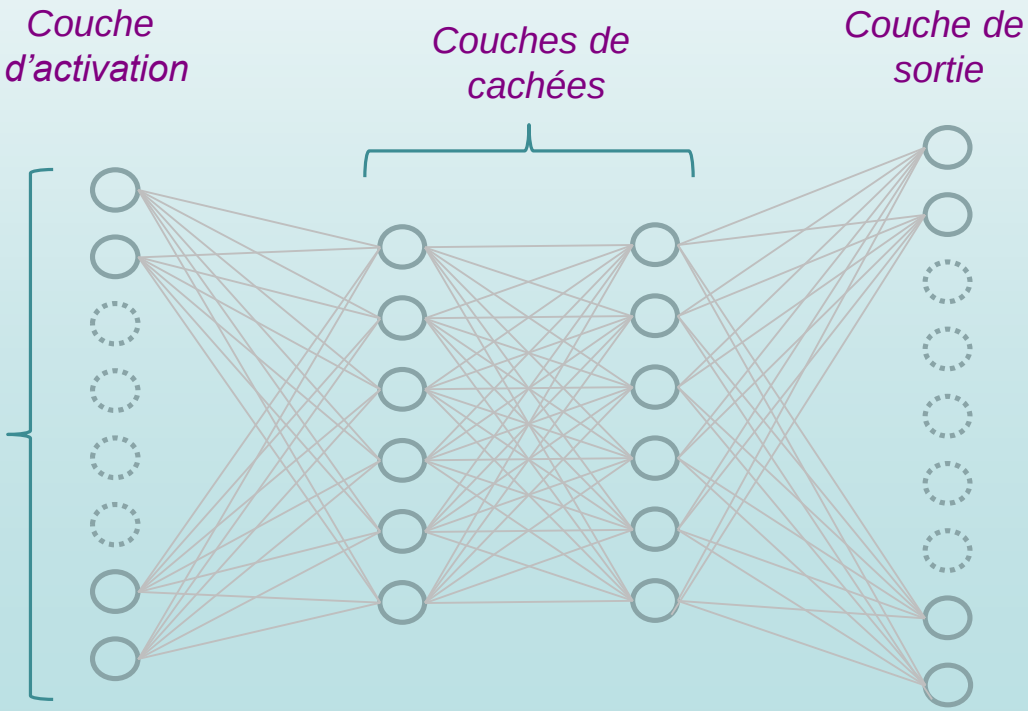
Réseaux de neurones formels

- *Sont constitués d'un grand nombre de neurones interconnectés, qui peuvent manipuler des valeurs binaires ou réelles.*
- *Le calcul de la sortie peut être déterministe ou probabiliste.*
- *L'architecture peut être définie avec ou sans rétroaction.*
- *La dynamique du réseau peut être synchrone ou asynchrone.*

Neurone : Perceptron

Réseaux de neurones

- *Un réseau de neurones est l'association, en un graphe plus ou moins complexe, d'objets élémentaires, les neurones formels.*
- *Un réseau se distingue par son architecture, le nombre et le type de neurones, les fonctions d'activation et les objectifs visés.*



Propagation des données de couches en couches jusqu'à la sortie

Pour un neurone j de la couche i

$$y_{i,j} = \theta\left(W_{I,J} \times X_{i-1} + b_{i,j}\right)$$

En cas de différence entre le résultat obtenu et la sortie espérée les poids W et les biais b sont mis à jour par rétro-propagation

Neurone : Perceptron

Le perceptron

- *Le perceptron est un neurone avec un algorithme d'apprentissage créé par Frank Rosenblatt en 1958.*
- *Il permet de prendre un décision à partir d'une fonction linéaire suivie d'une fonction de seuil (fonction d'activation).*
- *Un perceptron linéaire à seuil prend en entrée n valeurs [x_1 …x_n] et calcul une sortie y, définie par n synapses [w_1 …w_n] et un seuil θ.*

$$y = \theta(x) = \begin{cases} 1 & si\ f(x) = \sum_1^n w_i x_i > \theta \\ 0 & sinon \end{cases}$$

- *Les variables [w_1 …w_n] et le seuil θ sont les paramètres du modèle.*
- *Le modèle ajuste ces paramètres sur les données d'entrainement et sera en mesure pour une nouvelle valeur x de calculer la sortie associée.*
- *Un perceptron est adapté à la classification binaire.*

Neurone : Perceptron

Variantes

- *Les entrées peuvent être binaires ou réelles, les poids entiers ou réels. La sortie peut prendre ses valeurs dans {0,1}, il existe des modèles qui fournissent des sorties probabilistes.*
- *Afin de simplifier les calculs, une entrée x_0 (toujours égale à 1, appelée biais) de poids synaptique $w_0=-\theta$ est ajoutée.*
- *Dans ce cas si la fonction f(x) est positive la sortie est à 1 et à 0 sinon.*

Interprétation

- *Le perceptron linéaire à seuil à n entrées divise l'espace des entrées $\mathfrak{R}^n$ en deux sous espaces limités par un hyperplan.*
- *Par contre une fonction qui ne divise pas l'espace des entrées en deux hyperplans ne peut se représenter par un perceptron.*
- *Par exemple XOR ne peut être calculé par un perceptron à seuil.*
- *En effet, aucune droite ne peut séparer les points de cordonnées (0,0) et (1,1) des points de coordonnées (1,0) et (0,1).*

Neurone : Perceptron

Apprentissage par correction d'erreur

- *Chaque fois qu'un nouvel exemple est présenté on ajuste les poids selon que le perceptron l'a correctement classé ou non.*
- *L'algorithme s'arrête lorsque tous les exemples on été présentés sans qu'aucun poids n'est été modifié.*

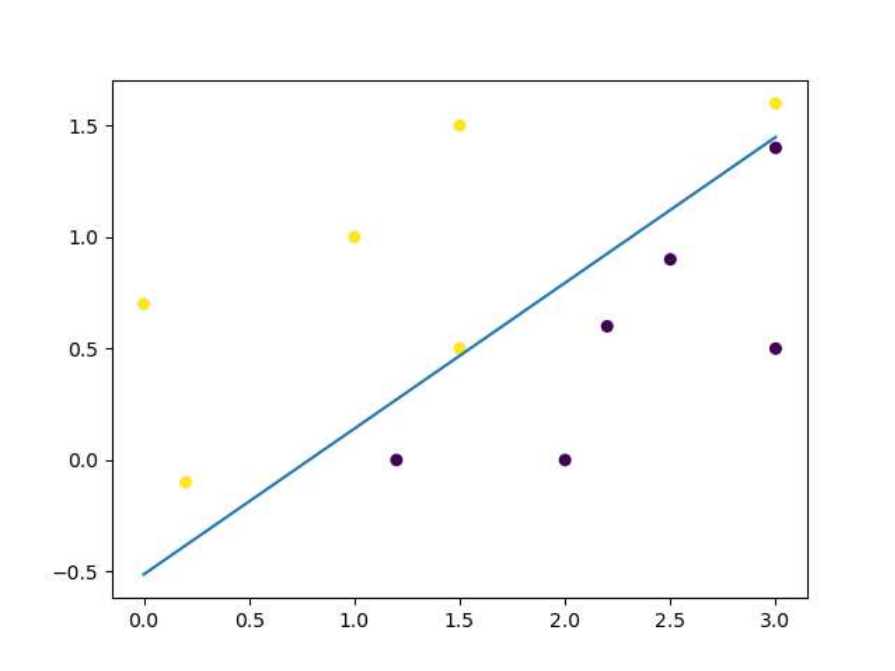
def *perceptron (Jeu de données* $L=[X^1 \ldots X^m]$*, sorties associées* $LY = [ly^1 \ldots ly^m]$*)*
génération aléatoire des poids synaptiques $W = [\theta, w_1, \ldots w_n]$
répéter :
foreach $(x, ly)^j \in (L,LY)$ *:*
calculer y^j
for $(i=0; i<n; i++) : w_i = w_i + (ly^j - y^j)\, x_i^j$
tant que *(le jeu de données L entraine des modifications sur W)*

- *Si* $y^j=0$ *et* $ly^j=1$*, cela signifie que le perceptron n'a pas assez pris en compte les dendrites, dans ce cas il faut ajuster les poids.*
- *Si* $y^j=1$ *et* $ly^j=0$*, il faut retrancher la valeur de la dendrite aux poids.*

Neurone : Perceptron

Exemple

- *On souhaite classer des données dépendant de deux features (x_1,x_2).*
- *Le jeu de données est parcouru plusieurs fois. Chaque parcours du jeu de données est appelé une époque (epoch).*
- *Il est recommandé de présenter les données de chaque époque dans un ordre aléatoire.*




```
Data = np.loadtxt('Perceptron.txt')
X = Data[:,:3] ; Ly = Data[:,3]
def Perceptron(epoch):
    indice = np.arange(len(Data))
    W = np.zeros(len(X[0]))
    for e in range(epoch):
        Wnext = W.copy()
        for i in indice :
            y = 0 ; if (np.dot(X[i],W))>=0 : y = 1
            W = W + (Ly[i]-y)*X[i]
        if (Wnext==W).all() : return W
        np.random.shuffle(indice)
    return W
```

Neurone : Perceptron

Inconvénients du perceptron

- *Si un échantillon est linéairement séparable et que les exemples sont présentés sans en exclure aucun, la procédure d'apprentissage par correction d'erreur converge vers un perceptron linéaire à seuil.*
- *Par contre si l'échantillon n'est pas linéairement séparable l'algorithme ne converge pas.*
- *Si pour un échantillon les poids prennent deux fois les mêmes valeurs sans que le perceptron n'ait appris, l'échantillon n'est pas séparable.*
- *L'inconvénient est que l'algorithme peut nécessiter un nombre exponentiel d'étapes avant de s'arrêter.*
- *De plus rien ne prouve que la solution obtenue est robuste, une entrée supplémentaire peut modifier la solution.*
- *Pire, l'algorithme n'a aucune tolérance au « bruit », c'est-à-dire qu'une information mal classée peut très facilement conduire à un échantillon non linéairement séparable.*

Neurone : Descente de gradient

Erreur d'un perceptron

- *Plutôt que de chercher un perceptron qui classifie correctement tous les exemples, on va essayer de trouver le minimum d'une fonction d'erreur.*
- *L'erreur d'un perceptron est défini comme la moyenne des pertes entre les sorties ly^j espérées pour les entrées x^j et les sorties calculées y^j.*
- *Pour une entrée la perte est donnée par :*

$$E^j(w) = -\left(ly^j - y^j\right)\sum_{i=0}^{n} w_i x_i^j$$

- *Si ly-y est positif cela implique que Σwx est inférieur à 0 c'est-à-dire que le calcul est inférieur à l'estimation, la perte sera donc positive.*
- *Cette perte deviendra négative dans le cas contraire et nulle si ly=y.*
- *L'apprentissage du perceptron consiste à minimiser la perte moyenne :*

$$E(w) = \frac{1}{m}\sum_{(X,ly)^j \in (L,LY)} -\left(ly^j - y^j\right)\sum_{i=0}^{n} w_i x_i^j$$

Neurone : Descente de gradient

Minimisation d'erreur

- *En machine learning ou en Outils et Méthodes de l'IA l'algorithme de descente du gradient est utilisé pour trouver le minimum des fonctions d'erreurs.*
- *Les variables à optimiser sont déplacées dans la direction opposée au gradient, afin de faire décroître la fonction d'erreur.*
- *Les poids W avancent dans le sens inverse des dérivées.*
- *On se doit de calculer toutes les dérivées de la fonction d'erreur E(W) dans la direction de tous les poids.*
- *On obtient alors la jacobienne de la fonction d'erreur : $\nabla E(W)$*
- *Il faut ensuite modifier les poids d'une fraction de $\nabla E(W)$.*

Initialisation des poids W
tant que W n'a pas convergé
$W^{k+1} = W^{k} - \alpha \nabla E(W)$

$$\nabla E(W) = \left[\frac{\partial E(w)}{\partial w_0}, \frac{\partial E(w)}{\partial w_1}, \dots, \frac{\partial E(w)}{\partial n}\right]$$

α est le taux d'apprentissage ou "Learning rate"

Neurone : Descente de gradient

Méthode du gradient

- *Le principe consiste à construire une suite récurrente $w^{k+1} = w^k + \Delta w^k = w^k - \alpha E'(w^k)$ qui à chaque itération conduit à avoir $E(w^{k+1}) < E(w^k)$.*
- *La difficulté ici est de bien choisir la valeur de α.*
- *Attention : rien ne garantit que la valeur trouvée est un minimum global.*
- *Le gradient de la perte moyenne par rapport aux différents poids :*

$$\frac{\partial E(w)}{\partial w_i} = \frac{1}{m} \sum_{(X,ly)^j \in (L,LY)} -\left(ly^j - y^j\right) x_i^j$$

- *Pour obtenir cette valeur il est nécessaire de calculer les sorties sur tous les exemples d'entrainement avant de faire les mises à jour des w_i.*
- *La descente de gradient stochastique est une variante plus efficace lorsque le corpus est important.*
- *Le principe consiste à ne sélectionner pour une époque qu'une sous-partie des données d'entrainement (miniEpoch) pour l'apprentissage.*

Neurone : Descente de gradient

Algorithme de descente de gradient

- *Le taux d'apprentissage α [0,1] doit être adapté au cas par cas.*
- *Avec un taux α/(1+βk) on aura une convergence même lorsque les valeurs ne sont pas linéairement séparables.*
- *k correspond au nombre d'erreurs commises, et β un hyperparamètre.*

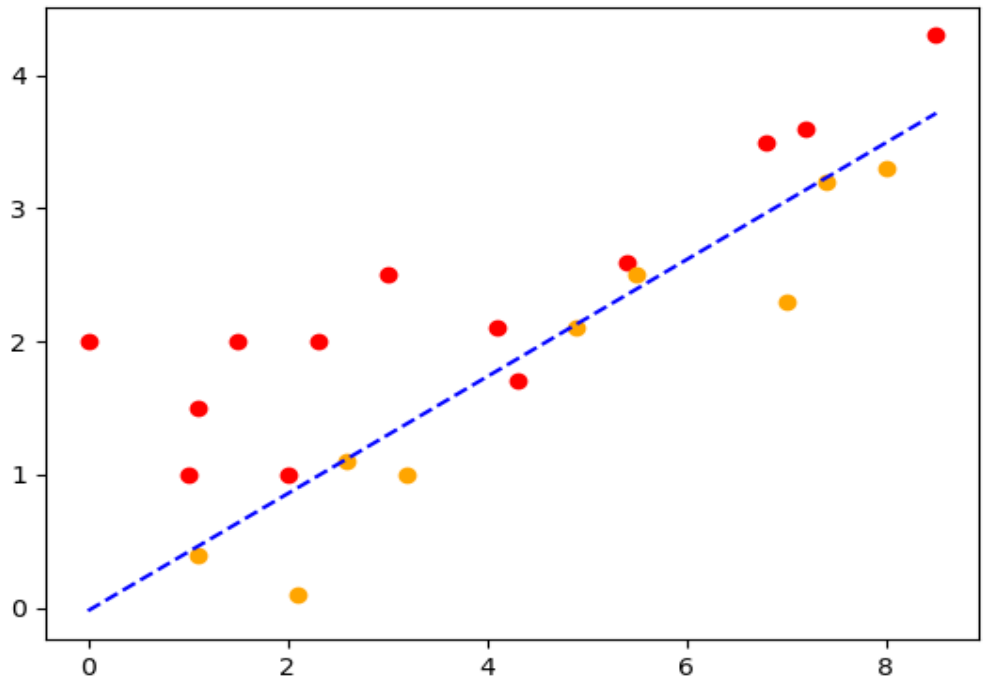
```
def descente_gradiant( (L,LY), α, ε=0.001, β=0.1, maxIter=5000) :
    génération aléatoire des poids synaptiques W' = [w_0, … w_n] ; k=0
    while :
        W=W'
        foreach (x, ly)^j ϵ (L,LY) : calculer y^j
            if (ly^j != y^j) : k+=1
                for (i=0; i<n; i++) : w_i' = w_i + (α (ly^j -y^j) x_i^j )/(1+ β*k)
    until ((∑_i^n (w_i' − w_i)^2) > ε)  and (iter<maxIter)
```

- *Dans ce cas l'algorithme converge vers une solution qui minimise le taux d'erreur sur les variables d'entrainement.*

Neurone : Descente de gradient

Convergence

- *Même si la série de points n'est pas linéairement séparable les algorithmes par correction d'erreur et du gradient permettent de trouver la solution.*



Avec une initialisation des W à 0, beta=0.1
La solution est obtenues en 520 itérations avec un k=134
W = [0.067 ; -1.281 ; 2.913]

Réseaux de neurones : MLP

- Fonctions d'activation
- Modèle d'apprentissage
- Rétropropagation
- Classement d'images

Réseaux de neurones : MLP

Perceptron multicouches

- *Un perceptron est un modèle très simple, capable de calculer les sorties comme des combinaisons linéaires des variables d'entrée plus un biais.*
- *Afin de couvrir un plus large éventail de modèles, les neurones peuvent être organisés en couches pour former un modèle plus complexe.*
- *Un perceptron multicouches (multilayer perceptron MLP) permet de trouver des solutions lorsque les jeux de données ne sont pas linéairement séparables.*
- *Un MLP est un réseau à couches cachées qui vérifie les propriétés :*
  - *Les neurones sont réparties dans des couches C_0, C_1, ..., C_m,*
  - *C_0 est composée des neurones d'entrée qui correspondent aux n variables ; C_1, ..., C_{m-1} sont les couches cachées ; C_m est composée du (ou des) neurone(s) de décision.*
  - *Les entrées d'un neurone de la couche C_i pour $i \geq 1$ appartiennent uniquement à la la couche C_{i-1} et à aucune autre couche.*
- *Un réseau MLP à couches cachées est sans rétroaction.*

MLP : Fonctions d'activation

Perceptron multicouches

- *Pour utiliser un MLP nous devons effectuer deux étapes :*
  - *Choisir l'architecture : définir le nombre de couches cachées et de neurones par couches .*
  - *Définir un algorithme d'apprentissage qui calcul à partir du jeux de données les valeurs des coefficients synaptiques.*
- *La première étape est encore aujourd'hui un sujet de recherche.*
- *L'algorithme de rétro-propagation du gradient proposé dans les années 80 permet de calculer les coefficients synaptiques du MLP.*

Fonction d'activation

- *Avec des perceptrons linéaires les algorithmes d'apprentissage ne peuvent pas converger. Ils sont alors instables*
- *La fonction de seuil doit être remplacée par une fonction d'activation.*
- *Elle joue un rôle central dans les réseaux de neurones car elle permet d'appliquer une transformation non linéaire aux données.*

MLP : Fonctions d'activation

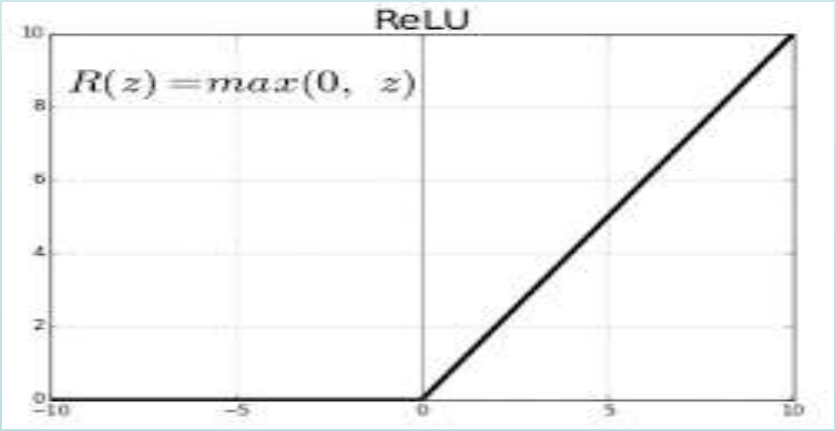
Les fonctions d'activation

- *Les fonctions d'activation non linéaires assurent une convergence des algorithmes d'apprentissage, comme la rétro-propagation, mais permettent également de modifier la représentation spatiale des données.*
- *Ces modifications sont un des atouts majeurs des RdN.*
- *Les RdN peuvent alors extraire des informations pertinentes lors de la phase d'apprentissage et produire des prédictions de qualité.*
- *La non-linéarité est une des caractéristiques qui différencie les algorithmes de Outils et Méthodes de l'IA de ceux du Machine Learning traditionnel.*
- *Il existe plusieurs fonctions d'activation, mais seulement une petite fraction d'entre elles est utilisée en Outils et Méthodes de l'IA.*
- *Les différences entre ces fonctions semblent négligeables, mais un mauvais choix peut impacter fortement les résultats des RdN.*
- *Les fonctions les plus utilisées sont : ReLU, Sigmoid, Softmax, Softplus, Tanh, ELU.*

MLP : Fonctions d'activation

ReLU (Rectified Linear Unit)

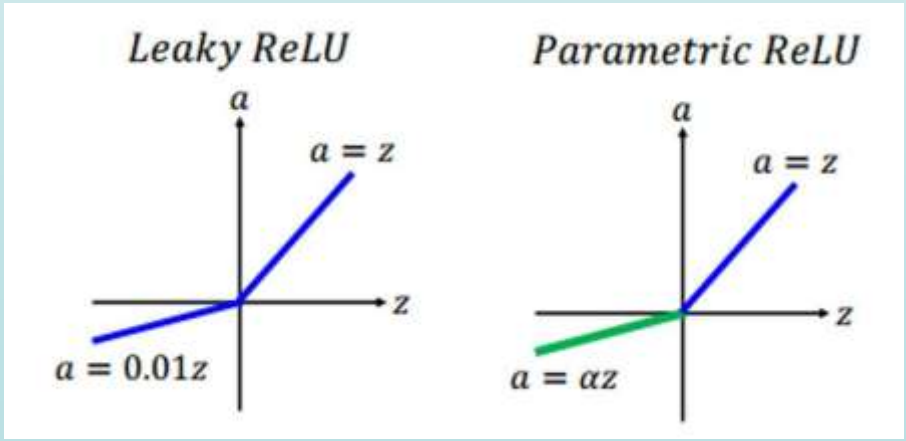
- *La fonction Rectified Linear Unit (ReLU) est la fonction d'activation la plus couramment utilisée en Outils et Méthodes de l'IA.*
- *Cette fonction qui applique un filtre en sortie permet au modèle de se concentrer uniquement sur certaines caractéristiques des données.*
- *La fonction ReLU permet de traiter le problème de l'évanouissement du gradient dans les réseaux neuronaux profonds.*
- *Ce problème survient lorsque le gradient devient trop faible au cours de l'étape de rétropropagation, ce qui rend l'apprentissage du réseau difficile.*

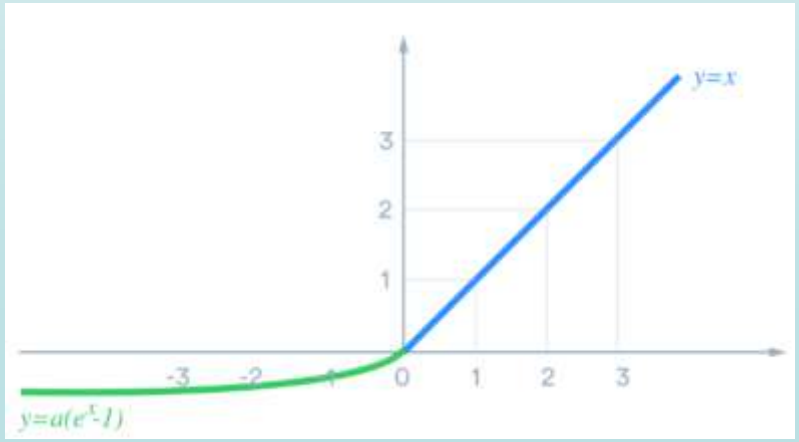


MLP : Fonctions d'activation

Les variantes de RELU :

- *La Leakey Relu permet d'ajouter une variante pour les nombres négatifs, ainsi les neurones ne meurent jamais.*
- *PReLU (Parametric ReLU) : le paramètre α est appris, cela permet au réseau de mieux s'adapter aux données et d'améliorer les performances.*
- *ELU (Exponential Linear Unit) : approche les valeurs moyenne proche de 0, afin d'améliorer les performances d'entrainements.*
- *ELU utilise une exponentiel pour la partie négative. Cette fonction peut être plus performante en expérimentation que les autres Relu.*





MLP : Fonctions d'activation

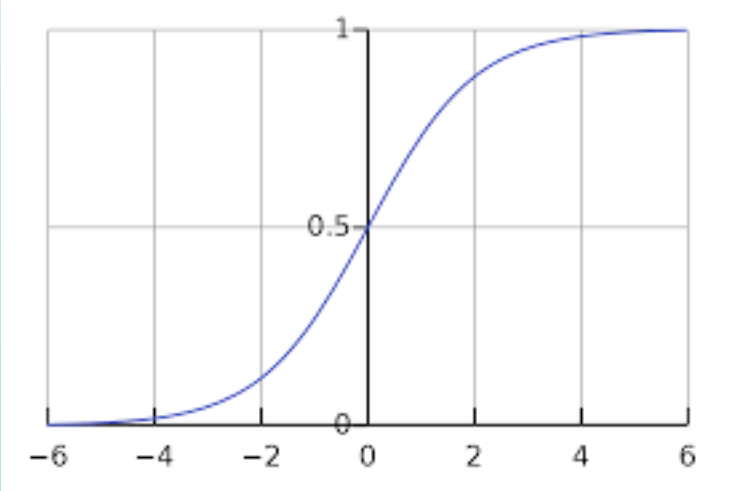
Sigmoid

- *La fonction Sigmoïd est utilisée en tant que dernière couche d'un réseau de neurones construit pour effectuer une tâche de classification binaire.*
- *Dans une classification binaire, une sigmoïde permet d'obtenir, pour une donnée, la probabilité d'appartenir à une classe donnée.*
- *La dérivée de sa fonction est extrêmement simple à calculer, ce qui permet d'améliorer les performances des algorithmes d'optimisation.*
- *Face à l'évanouissement du gradient dans les réseaux neuronaux profonds, la fonction sigmoid est moins efficace que Relu.*

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

et

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = \sigma(x)(1 - \quad (x))$$

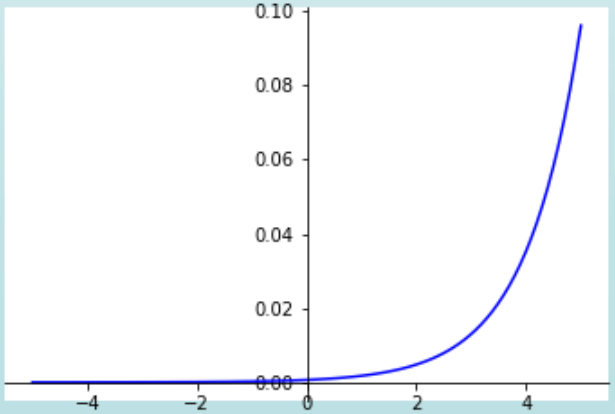


MLP : Fonctions d'activation

Softmax

- *La fonction Softmax est également utilisée en tant que dernière couche d'un RdN afin d'effectuer une tâche de classification multi-classes.*
- *Pour chaque sortie, Softmax donne un résultat entre 0 et 1. De plus, la somme de ces sorties est égal à 1.*
- *Les valeurs de la fonction peuvent être interprétées comme des probabilités. Chacune des valeurs est associée à une classe du dataset.*
- *La fonction Sigmoïd ne peut pas être utilisée en dernière couche du réseau pour une classification multi-classes, car les éléments additionnés pourraient ne pas être égaux à 1.*

$$\sigma(x_i) = \frac{\exp(x_i)}{\sum_i \exp(x_i)}$$

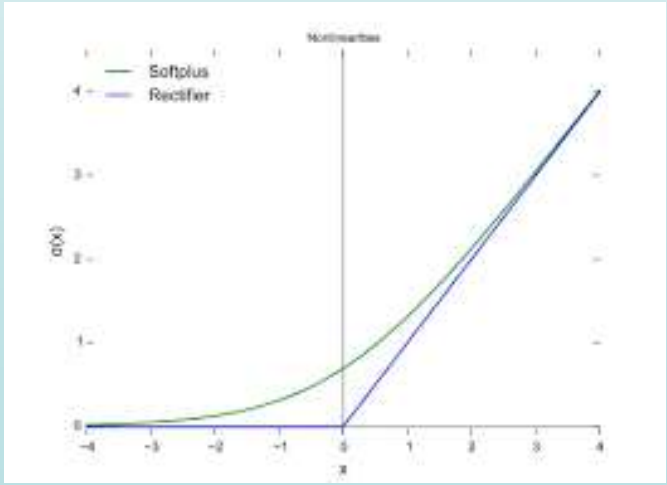


MLP : Fonctions d'activation

Softplus

- *Cette fonction est une approximation « lisse » de la fonction ReLU..*
- *Lorsque les valeurs d'entrée sont positives, la fonction Softplus se comporte, presque comme la fonction ReLU.*
- *Toutefois, pour des valeurs d'entrée négatives, la fonction Softplus n'applique pas de filtre comme ReLU, mais les fait tendre vers zéro.*
- *La courbe asymptotique autour de 0 permet d'obtenir des gradients plus stables lors de la retro-propagation ce qui peut améliorer l'entraînement du modèle.*

$$\sigma(x) = \log(\exp(x) + 1)$$



MLP : Fonctions d'activation

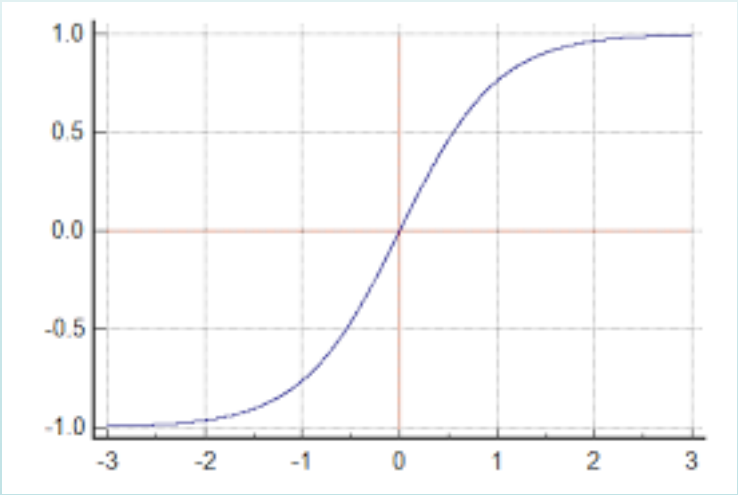
tanh

- *Cette fonction permet d'appliquer une normalisation aux valeurs.*
- *Elle peut également être utilisée au lieu de la fonction Sigmoid dans la dernière couche d'un modèle de classification binaire*
- *Elle donne un résultat entre -1 et 1.*

$$\sigma(x) = \frac{\sin(x)}{\cos(x)}$$

ou

$$\sigma(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$



MLP : Modèle d'apprentissage

Les réseaux à propagation avant avec Tensorflow

- *La structure d'un RdN artificiels concerne la multiplication matricielle.*
- *Les entrées des couches associées à un biais sont multipliées par des poids, le résultat est transformé par une fonction d'activation.*
- *Les valeurs de sortie sont utilisés pour faire une prédiction, et une mesure de distance permet d'estimer la capacité de prédiction du RdN.*
- *Tensorflow propose plusieurs façons de définir un réseau de neurones.*
- *La plus courante pour les réseaux à propagation avant (feedforward), qui empilent les couches, est d'utiliser la classe Sequential.*
- *La première étape consiste à cerner le problème en identifiant les données présentes dans le dataSet.*
- *On doit avant tout préciser la forme (shape) des données d'entrée.*
- *Ensuite on peut définir les différentes couches qui composent le RdN.*
- *Pour chaque couche il faut préciser le nombre de neurones, et la fonction d'activation à appliquer à tous les neurones.*

MLP : Modèle d'apprentissage

Les réseaux à propagation avant avec Tensorflow

- *En plus de la définition de l'architecture du réseau, il est nécessaire de définir comment seront mis à jour les poids des différents neurones.*
- *Afin d'entrainer le modèle (méthode **compile**) il est nécessaire de spécifier quelle est la méthode d'apprentissage à utiliser (optimizer).*
- *Il faut également préciser quelle est la fonction à minimiser (loss) ainsi que comment sera évaluer la qualité du modèle (metrics).*
- *Ces paramètres sont spécifiés lors de la phase de compilation du modèle à l'aide de la méthode **compile**.*
- *Les réseaux à propagation avant n'ont pas de mémoire. Cela signifie que les entrées et les sorties sont indépendantes.*
- *Ces RdN ne sont donc pas adaptés lorsque l'on doit prédire des séries chronologiques ou temporelles, c'est-à-dire lorsqu'une sortie dépend des informations passées.*
- *Par exemple pour prédire des phrases, il est nécessaire de connaitre le contexte et donc d'utiliser un autre type de réseau de neurones.*

MLP : Modèle d'apprentissage

Choix de la fonction d'optimisation : optimizer

- *La **descente de gradient** (Gradient Descent) est une technique rapide, efficace, robuste, flexible très utilisée pour le Outils et Méthodes de l'IA.*
- *Cette technique a été déclinée en différentes versions : descente de gradient par batch, stochastique (SGD), mini-batch.*
- ***Adam** (Adaptative Momentum estimation) le plus utilisé pour les réseaux de neurones, en raison de son efficacité et de sa stabilité.*
- *Le principe consiste à accélérer la vitesse d'apprentissage tant que le gradient est dans la même direction que les précédents.*
- *L'optimiseur **RAdam** est une évolution d'Adam, elle corrige sa principale faiblesse à savoir la forte dépendance en la vitesse d'apprentissage.*
- *En ne dépendant pas de cet hyper-paramètre l'algorithme converge vers la même précision quelque soit le learning rate.*
- ***RMSprop** accélère la descente de gradient, il est utilisé pour entraîner des modèles à base de réseaux de neurones profonds.*

MLP : Modèle d'apprentissage

Choix de la fonction d'erreur : loss

- *La fonction d'erreur (loss) est une des composantes essentielles qui permet d'entraîner un modèle.*
- *Cependant, il y existe plusieurs manières de calculer l'erreur entre les prédictions et les valeurs réelles. Le choix dépend du modèle.*
- *Classification binaire : la dernière couche doit posséder une fonction d'activation sigmoïde, la **Binary Cross Entropy** est alors la plus utilisée.*
- *La **Cross Entropy** est également utilisée en classification multi-classes.*
- *La **Hinge Loss** est une alternative lorsque les données sont complexes. La couche de sortie doit avoir une activation de type tanh.*
- *La **Kullback-Leibler Divergence** (KL Divergence) est utilisée pour la classification mais aussi pour des tâches de générations.*
- *La **Mean Squared Error** (MSE), est une des fonctions de perte les plus utilisée dans les problèmes de régression.*
- *La **Huber Loss** est particulièrement utile lorsque le dataset contient beaucoup d'outliers.*

MLP : Modèle d'apprentissage

Qualité de l'optimiseur : metric

- *La matrice de confusion ou matrice d'erreur est un tableau qui permet de visualiser les performances d'un algorithme d'apprentissage supervisé.*
- *Comme souvent il y a plus de valeurs d'une classe que d'une autre, une étude directe va biaiser les résultats vers la valeur la plus haute.*

$valeurs\ réelles$ →

$valeurs\ prédites$ ↕

	Positifs	Négatifs	
Positifs	Vrais-Positifs	Faux-Positifs	Recall VP/(VP+FP)
Négatifs	Faux-Négatifs	Vrais-Négatifs	Sensitivity VN/(FN+VN)
	Precision VP/(VP+FN)	Specificity VN/(FP+VN)	Accuracy VP+VN/(total)

Si nous avons plus de deux classes à prédire – trois (A,B,C) il faut faire trois tableau A / (B+C) ; B /(A+C) et C / (A+B)

$$Score\ F1 = 2 * \frac{Precision(i) * Recall(i)}{Precision(i) + Recall(i)}$$

MLP : Modèle d'apprentissage

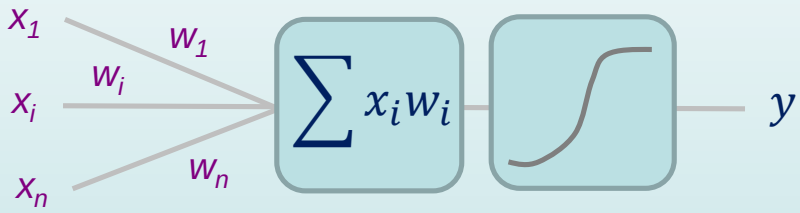
Qualité de l'optimiseur : metric

- *Les métriques sont similaires aux fonctions de perte, mais elles ne sont pas utilisées lors de la formation du modèle.*
- *L'**Accuracy** est la mesure la plus simple elle estime combien de fois le modèle a prédit correctement la target. Cette métrique est la plus utilisée.*
- *Plusieurs déclinaison de cette métrique existe : **BinaryAccuracy**, **CategoricalAccuracy** pour une prédiction de type one-hot, ….*
- *Les métriques de types probabilités. **BinaryCrossentropy** est une métrique d'entropie utilisée lorsqu'il n'y a que deux classes d'étiquettes.*
- ***SparseCategoricalCrossentropy** utilisée pour l'étiquetage "d'entiers".*
- *Les métriques de régression : **MeanSquaredError** mesure l'erreur quadratique moyenne entre les sorties calculées et les sorties attendues.*
- *Les métriques basées sur les classifications de type Vraie/Faux. L'**AUC** (Area Under Curve) décrit la performance d'un modèle de classification à travers deux indicateurs : la sensitivity et la specificity.*
- ***F1Score** métrique calculant le score F1 d'un modèle de classification.*

MLP : Retro-propagation

Modèles avec régression logistique

- *Dans la plupart des cas les fonctions d'activation σ sont non linéaires.*
- *Pour un modèle qui doit prédire en sortie une probabilité d'appartenance à la classe (0 ou 1) la fonction d'activation sigmoïde est utilisée :*



$$y = \sigma(f(x)) = \frac{1}{1 + e^{-f(x)}}$$

- *Nous supposerons que tous les neurones utilisent la même fonction σ.*
- *Si le réseau à p sorties est entrainé par le jeux de données (L,LY), à chaque présentation d'une donnée (x, Ly) $\in$ (L,LY) on a l'erreur :*

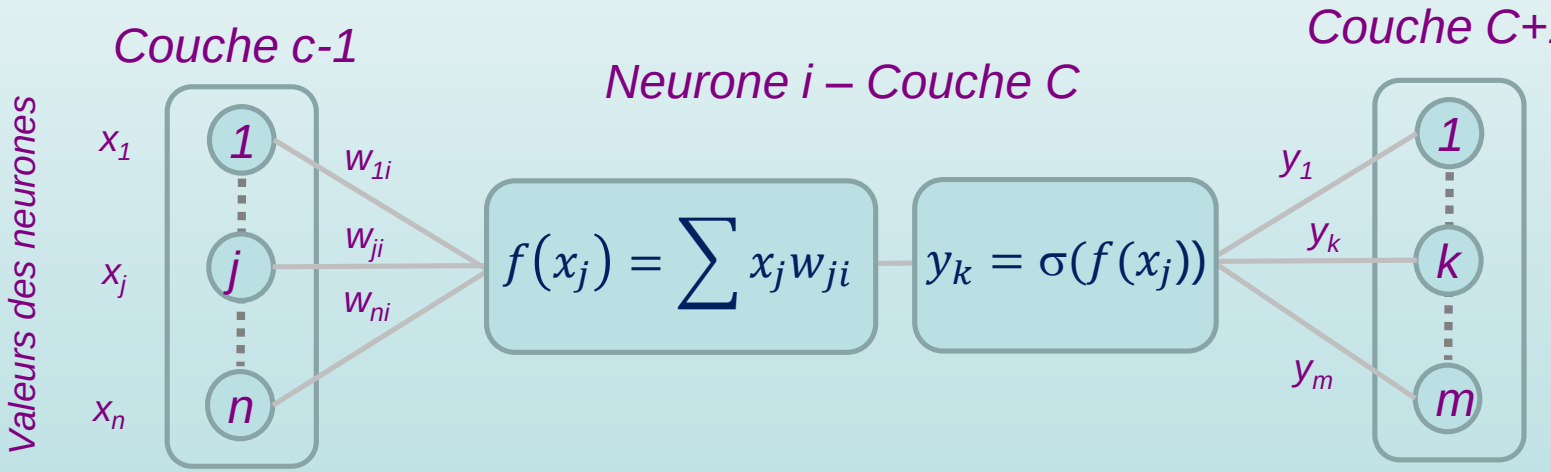
$$E(w) = \frac{1}{2}\sum_{k=1}^{p}(ly_k - y_k)^2$$

- *L'algorithme de rétro-propagation consiste à minimiser la fonction E à chaque présentation d'un entrée x sur la couche d'entrée.*

MLP : Retro-propagation

Algorithme de retro-propagation

- *La dérivée de E(w) par rapport aux différents poids nous permet d'évaluer la contribution de ces poids sur l'erreur.*
- *Un neurone i calculera sa sortie à partir des entrées notées x_i qui proviennent de la couche précédente, et fournira en sortie des valeurs qui serviront à alimenter les neurones de la couche suivante.*



- *Un neurone i de la couche c est influencé par tous les neurones de la couche précédente et impacte tous les neurones de la couche suivante.*

MLP : Retro-propagation

Dérivée de la fonction d'erreur

- *Nous devons évaluer pour tous les neurones i les valeurs de $\partial E/\partial w_{ji}$.*
- *Le poids w_{ji} influence la sortie du réseau au travers de la fonction y_i du neurone i et donc des neurones qui se trouve sur les couches suivantes.*

$$\frac{\partial E(W)}{\partial w_{ji}} = \frac{\partial E(W)}{\partial f(x_i)} * \frac{\partial f(x_i)}{\partial w_{ji}} = \frac{\partial E(W)}{\partial f(x_i)} x_{ji}$$

- *Pour le calcule de $\partial E/\partial f(xi)$ deux cas se présentent.*
- *Si i est un neurone de sortie alors.*

$$\frac{\partial E(W)}{\partial f(x_i)} = \frac{\partial E(W)}{\partial y_i} * \frac{\partial y_i}{\partial f(x_i)} \begin{cases} \frac{\partial E(W)}{\partial y_i} = \frac{\partial}{\partial y_i} \frac{1}{2} \sum_{k=1}^{p} (ly_k - y_k)^2 = (y_i - ly_i) \\ \frac{\partial y_i}{\partial f(x_i)} = \sigma\big(f(x_i)\big)\Big(1 - \sigma\big(f(x_i)\big)\Big) = y_i(1 - y_i) \end{cases}$$

- *On en déduit alors la dérivée* $\frac{\partial E(W)}{\partial w_{ji}} = y_i(y_i - ly_i)\,(1 - y_i)x_{ji}$

MLP : Retro-propagation

Dérivée de la fonction d'erreur

- *Si i n'est pas une sortie, y_i va alors être influencé par ses successeurs.*

$$\frac{\partial E(W)}{\partial f(x_i)} = \sum_{k \in succ(i)}^{m} \frac{\partial E(W)}{\partial f(x_k)} * \frac{\partial f(x_k)}{\partial f(x_i)} = \sum_{k \in succ(i)}^{m} \frac{\partial E(W)}{\partial f(x_k)} * \frac{\partial f(x_k)}{\partial y_i} * \frac{\partial y_i}{\partial f(x_i)}$$

$$\frac{\partial E(W)}{\partial f(x_i)} = \sum_{k \in succ(i)}^{m} \frac{\partial E(W)}{\partial f(x_k)} w_{ki}\, y_i(1 - y_i) = y_i(1 - y_i) \sum_{k \in succ(i)}^{m} \frac{\partial E(W)}{\partial f(x_k)} w_{ki}$$

- *Le calcul des différentes dérivées se fera en remontant les couches depuis la couche de sortie jusqu'aux neurones de la première couche.*

Modification des poids synaptiques

- *On en déduit alors la nouvelle variable de w_{ji} par la méthode du gradient*

$$w_{ji} = w_{ji} - \alpha \frac{\partial E(W)}{\partial f(x_i)} x_{ji}$$

MLP : Retro-propagation

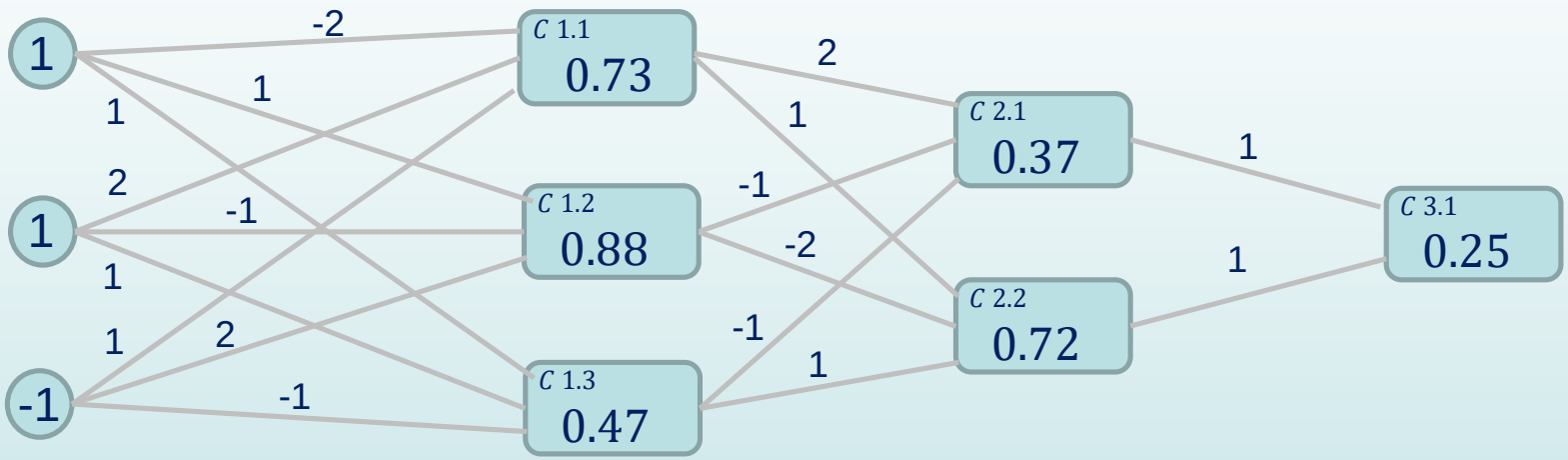
Algorithme de rétro-propagation du gradient

- $\partial p(x_i)$ est la dérivée particulaire de l'erreur par rapport $f(x_i)$.

def *((L,LY), un MLP à q couches et nc neurones, ε) :*
génération aléatoire entre {-0.5,0.5] des poids de tous les neurones (w_{ji})
while :
$W' = W$ *soit* $[w'_1, \dots w'_n]=[w_1, \dots w_n]$
foreach $((x_0, ly)$ in (L,LY)) :
calculer $y=MLP(W,x_0)$ *// Valeur de sortie du perceptron pour* x_0 et W
foreach *(neurone* $i \in$ *couche de sortie) :* $\partial p(x_i)= y_i (1- y_i) (y_i - ly)$
for $(c=q-1; c>0; c--)$: *// pour chaque couche intermédiaire c*
foreach *(neurone* $i \in$ *couche c) :*
$\partial p(x_i) = y_i (1- y_i) \sum_{k \in succ(i)}^{n} \partial p(x_k) w_{ki}$
for $(i=1; i<nc; i++)$:
foreach (*j* ***in*** *dendrite de neurone(i)) :*
$w_{ji} = w_{ji} + \alpha \partial p(x_i) x_{ji}$
until : $(\sum_i^n (w'_i - w_i)^2) > \varepsilon$

MLP : Retro-propagation

Exemple : (x,fy) = ((1,1,-1), 1)

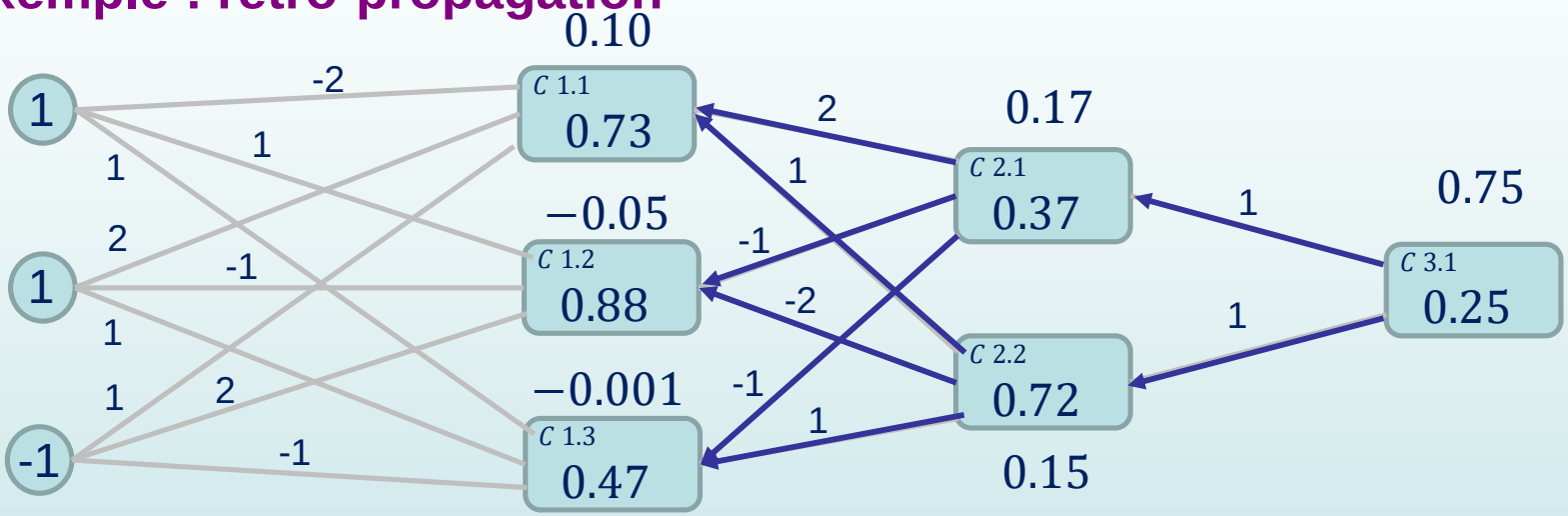


Phase 1 : Calcul des valeurs de sortie de MLP

*C 1.1 : $f(x)$=(-1*2 + 1*2 - 1*1)=-1 et $y_{c1.1} = \sigma(f(x))$=1/(1+ e^{-1}) = 0.73*
*C 2.1 : $f(x)$=(0.73*2 - 0.88*1 – 0.47*1)=0.53 et $y_{c2.1} = \sigma(f(x))$=1/(1+ $e^{-0.53}$) = 0.37*
*C 3.1 : $f(x)$=(0.37*1 + 0.72*1)=1.10 et $y_{c3.1} = \sigma(f(x))$=1/(1+ $e^{-1.10}$) = 0.25*
*C 2.2 : $f(x)$=(-0.73*1 - 0.88*2 + 0.47*1)=-0.98 et $y_{c2.2} = \sigma(f(x))$=1/(1+ $e^{-0.98}$) = 0.72*
*C 1.3 : $f(x)$=(1*1 + 1*1 + 1*1)=3 et $y_{c1.3} = \sigma(f(x))$=1/(1+ e^{3}) = 0.47*

MLP : Retro-propagation

Exemple : rétro-propagation



Phase 2 : Rétro-propagation – calcul dérivées $\partial p(x_i) = \partial E(W)/\partial f(x_i)$

C 1.1 : $y_i(1-y_i)\sum_{k\in succ(i)}^{2}\frac{\partial E(W)}{\partial f(x_k)}w_{ki}$ *= 0.73*(1-0.73)(2*0.17 +1*0.15)= 0.1 (i=1.1)*

C 2.1 : $y_i(1-y_i)\frac{\partial E(W)}{\partial f(x_k)}w_{ki}$ *= 0.37*(1-0.37)*1*0.75 = 0.17 (i=2.1 et k=3.1)*

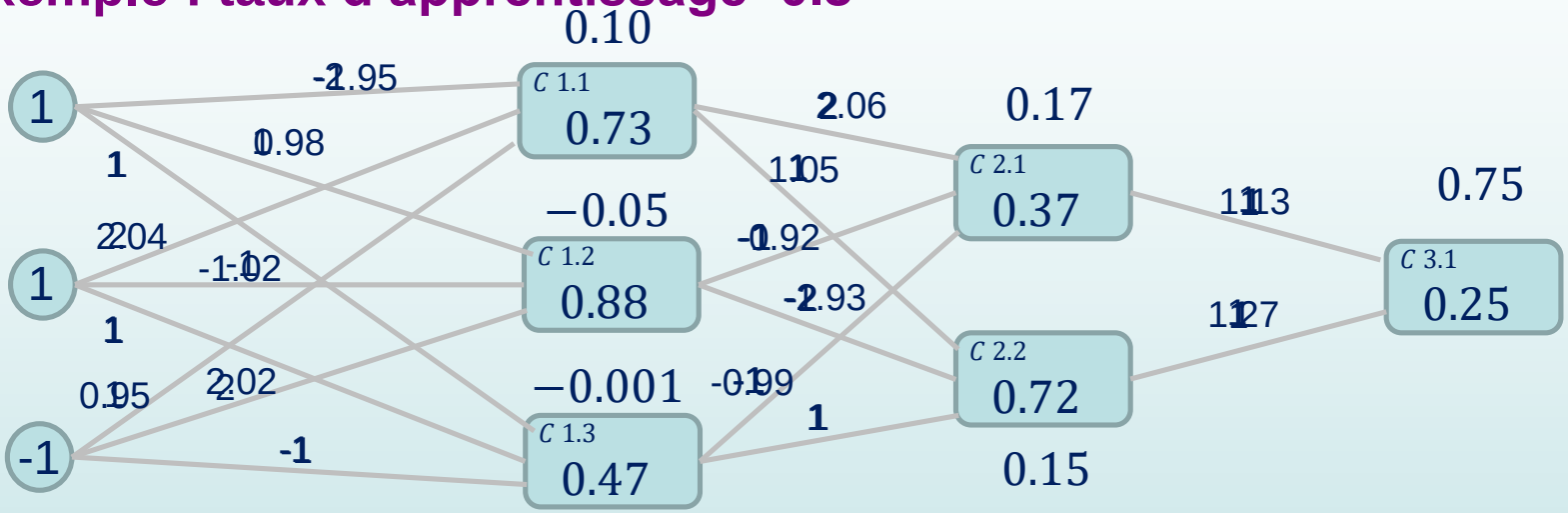
C 1.2 : $y_i(1-y_i)\sum_{k\in succ(i)}^{2}\frac{\partial E(W)}{\partial f(x_k)}w_{ki}$ *= 0.88*(1-0.88)(-1*0.17 -2*0.15)= -0.05 (i=1.2)*

C 2.2 : $y_i(1-y_i)\frac{\partial E(W)}{\partial f(x_k)}w_{ki}$ *= 0.72*(1-0.72)*1*0.75 = 0.15 (i=2.2 et k=3.1)*

C 1.3 : $y_i(1-y_i)\sum_{k\in succ(i)}^{2}\frac{\partial E(W)}{\partial f(x_k)}w_{ki}$ *= 0.47*(1-0.47)(-1*0.17 +1*0.15)= 0.001 (i=1.3)*

MLP : Retro-propagation

Exemple : taux d'apprentissage=0.5



Phase 3 : Mise à jour de poids

C 1.1 : $w_{1i} = w_{1i} + \alpha\, \partial p(x_1)\, x_{1i} = -2 + 0.5 * 0.1 * 1 = -1.95$

: $w_{2i} = w_{2i} + \alpha\, \partial p(x_2)\, x_{2i} = 2 + 0.5 * 0.1 * 1 = 2.04$

C 2.1 : $w_{1i} = w_{1i} + \alpha\, \partial p(x_1)\, x_{1i} = 2 + 0.5 * 0.17 * 0.73 = 2.06$

: $w_{3i} = w_{3i} + \alpha\, \partial p(x_3)\, x_{3i} = 1 - 0.5 * 0.1 * 1 = 0.95$

: $w_{2i} = w_{2i} + \alpha\, \partial p(x_2)\, x_{2i} = -1 + 0.5 * 0.17 * 0.88 = -0.92$

C 3.1 : $w_{1i} = w_{1i} + \alpha\, \partial p(x_1)\, x_{1i} = 1 + 0.5 * 0.05 * 0.37 = 1.13$

: $w_{3i} = w_{3i} + \alpha\, \partial p(x_3)\, x_{3i} = -1 + 0.5 * 0.17 * 0.47 = -0.99$

: $w_{2i} = w_{2i} + \alpha\, \partial p(x_2)\, x_{2i} = 1 + 0.5 * 0.05 * 0.72 = 1.07$

C 2.2 : $w_{1i} = w_{1i} + \alpha\, \partial p(x_1)\, x_{1i} = 1 + 0.5 * 0.15 * 0.73 = 1.05$

: $w_{3i} = w_{3i} + \alpha\, \partial p(x_3)\, x_{3i} = 2 + 0.5 * 0.05 * 1 = 2.02$

: $w_{2i} = w_{2i} + \alpha\, \partial p(x_2)\, x_{2i} = -2 + 0.5 * 0.15 * 0.88 = -1.93$

C 1.3 : $w_{1i} = w_{1i} + \alpha\, \partial p(x_1)\, x_{1i} = 1 - 0.5 * 0.001 * 1 = 1$

: $w_{3i} = w_{3i} + \alpha\, \partial p(x_3)\, x_{3i} = 1 + 0.5 * 0.15 * 0.47 = 1.00$

: $w_{2i} = w_{2i} + \alpha\, \partial p(x_2)\, x_{2i} = 1 - 0.5 * 0.001 * 1 = 1$

: $w_{3i} = w_{3i} + \alpha\, \partial p(x_3)\, x_{3i} = -1 + 0.5 * 0.001 * 1 = -1$

MLP : Modèle d'apprentissage

Tensorflow.keras : Les layers

- *Keras est une surcouche de TensorFlow qui contient des fonctionnalités de base telles que les couches, des opérations matricielles …*
- *Keras propose plusieurs types de couches : couches de base, couches à mémoire, d'activation, de normalisation, de calcul de perte, et d'optimisation.*
- *Couches de base :*
  - *Dense : Couche entièrement connectée*
  - *Conv1D, Conv2D, Conv3D : Couche de convolution*
  - *MaxPooling2D, MaxPooling2D : Couche de sous-échantillionage*
  - *Flatten : Aplatit les matrices en vecteurs unidimensionnels*
  - *InputLayer : Couche d'entré du réseau*
  - *Embedding : Couche de conversion de données discrètes (ex : mots) en vecteurs continus.*
  - *Dropout : Couche de régularisation par désactivation de neurones.*

MLP : Modèle d'apprentissage

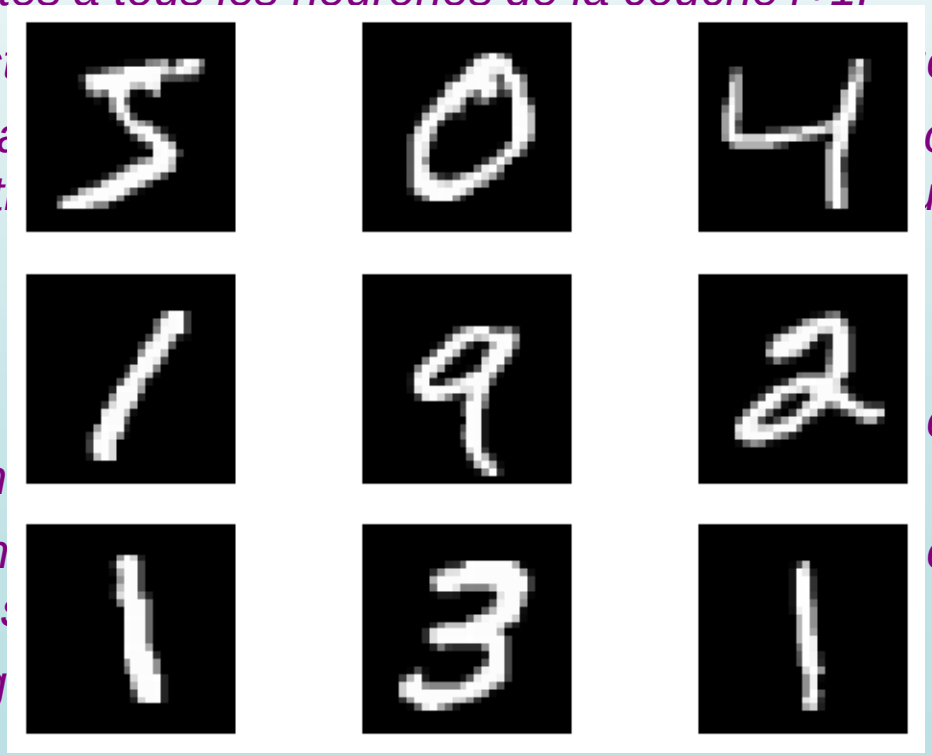
Tensorflow.keras : Les layers

- *Couches a mémoire :*
  - *LSTM, GRU : Couche avec mémoire*
  - *Attention : Couche utilisée pour gérer l'attention*
- *Couches de normalisation :*
  - *BatchNormalisation : Normalisation pour chaque batch*
  - *LayerNormalisation : Normalisation au niveau de la couche.*
- *Couches d'activation :*
  - *RelU, LeakyReLU, Sigmoid, Tanh, Softmax*
- *Couches de perte :*
  - *MeansSquaredError : Régressions*
  - *BinaryCrossEntropie, CatégoricalCrossEntropy : Classifications.*
- *Couches d'optimisation :*
  - *SGD, Adam : Descente de gradient*
  - *RMSprop : Optimisation pour des réseaux récurrents.*

MLP : Classement d'images

Identification de 'digits' - MLP

- *L'objectif est de développer un perceptron multicouches pour la classification de chiffres manuscrits issus de la base de données MNIST.*
- *Les MLP sont des réseaux entièrement connectés, tous les neurones d'une couche i sont connectés à tous les neurones de la couche i+1.*
- *Compte tenu de leur struct* *e.*
- *Ils utilisent la rétropropaga* *ont généralement utilisés pour t* *ue.*



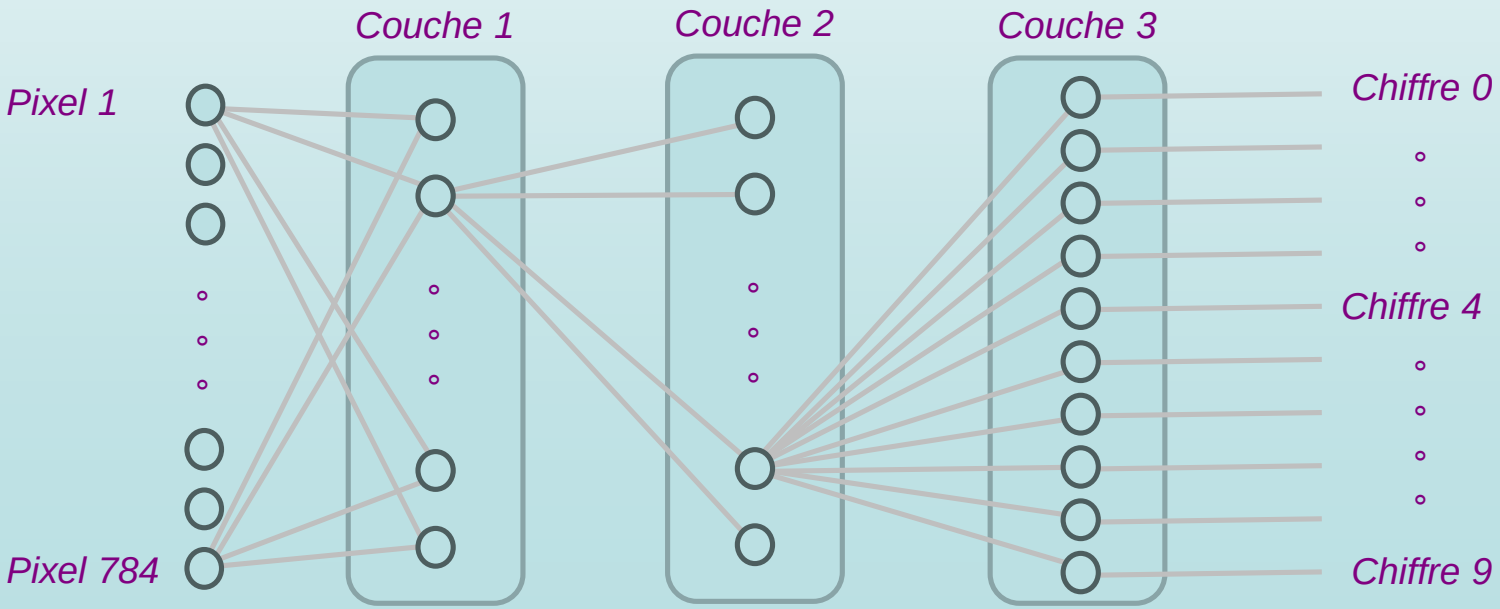
MNIST Digits

- *MNIST (Mixed National* *est une base de données de ch*
- *Cette base est suffisamn* *est très utilisée pour l'apprentis*
- *Les chiffres sont des imag*

MLP : Classement d'images

Les MLP avec Keras et Tensorflow

- *En entrée d'un réseau de neurone les images (28,28) doivent être redimensionnées en vecteur de taille 784.*
- *Le réseau de sortie est composé de 10 neurones, chaque neurone retourna une probabilité correspondant à chacun des 10 chiffres.*
- *Pour être conforme à la couche de sortie les targets sont codés one-hot. Exemple 2 = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]*



MLP : Classement d'images

Préparation des données

- *Toutes les fonctions peuvent être utilisées via soit le bibliothèque keras soit via la bibliothèque tensorflow.keras.*
- *Les données 'mnist' sont disponibles dans les datasets de keras.*
- *Les données d'entrée doivent être aplaties et normalisées. Les targets elles transformées en liste 'one-hot'.*

```
#reformatage des données d'apprentissage
vector_size= X_train.shape[1] * X_train.shape[2]
X_train = np.reshape(X_train, [-1, vector_size])
X_train = X_train.astype('float32')/256
X_test = np.reshape(X_test, [-1, vector_size])
X_test = X_test.astype('float32')/256
```

```
#mise en forme des targets
Y_train = keras.utils.to_categorical(Y_train)
Y_test = keras.utils.to_categorical(Y_test)
```

Construction du réseau

- *La classe **Sequential** permet de définir l'enchaînement des couches et des actions du réseau.*
- ***Dense** est une couche keras entièrement connectée à la précédente.*

MLP : Classement d'images

Construction du réseau

- *La couche d'entrée est composée de 784 neurones (1 par pixel).*
- *La couche de sortie de 10 neurones en charge de chaque chiffre.*
- *Le nombre de couches cachées et les fct d'activation sont à définir.*
- *Ces valeurs dépendent du problème étudié. Il n'y a aucune technique permettant de les estimer, elles restent à l'appréciation du développeur.*
- *Avec peu de neurones cachés, le réseau risque de ne pas converger, avec trop de neurones cachés, le risque est le surapprentissage.*
- *Nous proposons d'utiliser deux couches cachées, avec des fonctions d'activation de type 'relu' et de type 'softmax' pour la couche de sortie.*
- *Pour le nombre de neurones (nu_hidden) des couches cachées nous allons effectuer plusieurs tests avec un nombre de neurones égale à 2^n.*

```
#Création d'un modèle Séquentiel : MLP
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(nu_hidden, input_dim=784, activation='relu'))
model.add(tf.keras.layers.Dense(nu_hidden, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

MLP : Classement d'images

Modèle d'apprentissage : compile

- *Avant de pouvoir entrainer le réseau il faut définir quel modèle d'apprentissage utiliser pour régler les poids des neurones.*
- *On se doit de préciser la méthode d'apprentissage (optimizer), la fonction d'erreur à minimiser (loss) et enfin le score (metrics).*

```
#Exemple de modèle d'apprentissage
model.compile( loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Apprentissage : fit

- *Pour l'apprentissage il est important de définir le nombre d'époques.*
- *La taille des lots (batch_size) correspondant aux découpes de chaque époque. A la fin de chaque lots les poids des neurones seront mis à jour.*
- *Il est également souhaitable d'utiliser une partie des données du 'train' comme données de validation afin d'éviter le sur-apprentissage du MLP.*

```
#Exemple de fonction d'apprentissage
model.fit(X_train, y_train, epochs=20, batch_size=128, validation_split=0.2)
```

MLP : Classement d'images

Résultats

- *Résultats des tests nu_hidden=[8, 16, 32, 64, 128, 256]*

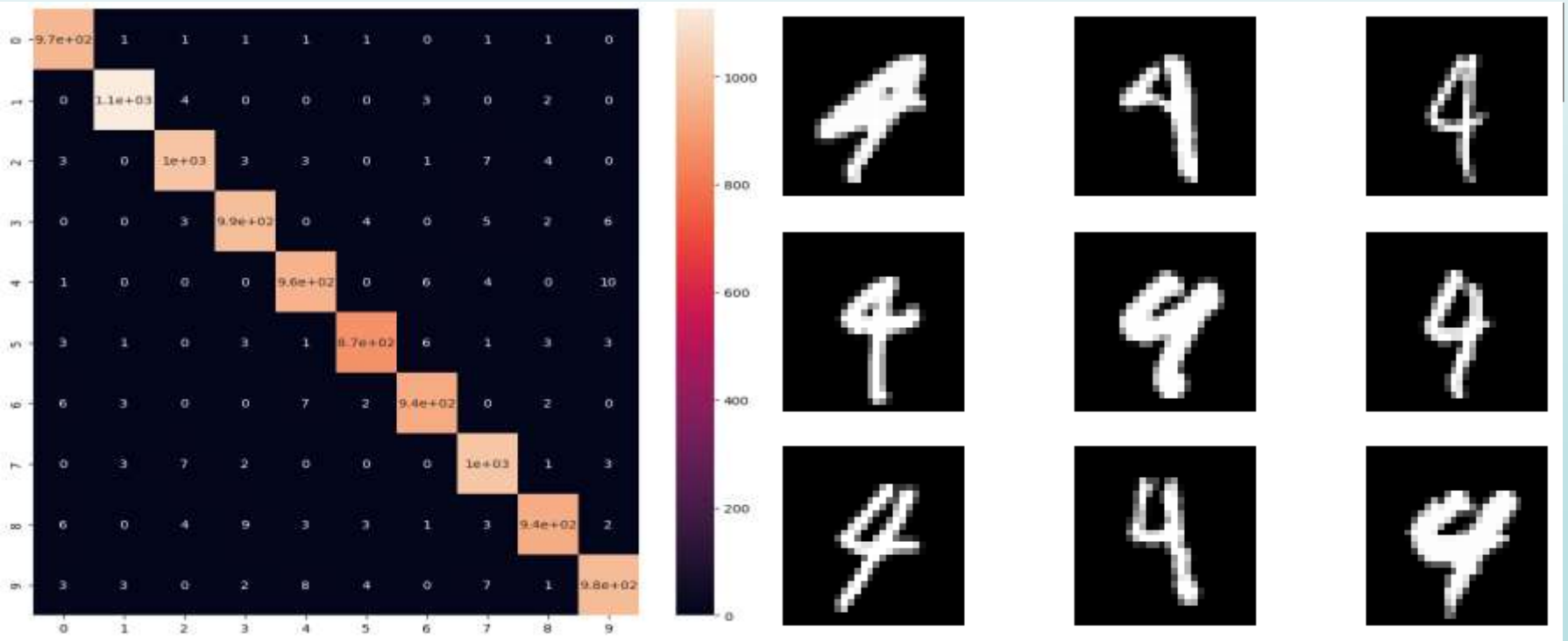
nu_hidden	loss	Accu	loss_val	Accu_val	Erreurs
8	0,253	0,928	0,261	0,929	761
16	0,121	0,964	0,164	0,953	487
32	0,047	0,986	0,132	0,964	325
64	0,012	0,997	0,135	0,970	285
128	0,005	0,999	0,107	0,978	211
256	0,003	0,999	0,134	0,977	208

- *A partir de deux couches cachées de 128 neurones les modèles sont clairement en sur-apprentissage.*
- *L'erreur est nulle, l'apprentissage est de 100%, mais les erreurs restent à 2% aussi bien sur les données de validation que de test.*
- *L'overfitting est fréquent lors de l'entraînement d'un modèle de Outils et Méthodes de l'IA, mais une technique existe pour le contrer : le **Dropout**.*
- *Le principe consiste à désactiver temporairement certains neurones dans le réseau, ainsi que toutes ses connexions entrantes et sortantes.*
- *Le choix des neurones à désactiver est aléatoire.*

MLP : Classement d'images

Résultat final

- *On attribue une probabilité p à tous les neurones qui détermine leur fréquence de désactivation.*
- *le Dropout est actif uniquement durant l'entraînement du modèle.*
- *L'instruction doit être ajouté après une ou les deux couches cachées.*



Réseaux de convolution : CNN

- Les features
- Les couches
- Exemple Quick-Draw

Réseaux de convolution : CNN

Réseaux de neurones convolutifs

- *Les CNN sont utilisés pour les tâches de classification de textes et d'images. Ils sont aussi efficaces pour le traitement du langage naturel.*
- *La reconnaissance d'image consiste à extraire des caractéristiques (features) en utilisant une série de transformations mathématiques, telles que des filtrages, des histogrammes ou des orientations de gradient.*
- *Les features telles que les coins, les bords ou les points d'intérêts, permettent d'identifier ou de reconnaitre une image.*
- *Depuis les années 2010, la majorité des algorithmes de vision reposent sur des caractéristiques qui sont apprises par des réseaux convolutifs.*
- *Les CNN réalisent eux-mêmes la travail d'extraction et de description des features : lors de la phase d'apprentissage.*
- *L'architecture spécifique des CNN permet d'extraire des features de différentes complexités, des plus simples aux plus sophistiquées.*
- *L'extraction des features est automatique, et s'adapte au problème donné, il n'est plus nécessaire d'implémenter les algorithmes.*

CNN : Les features

Les convolutions

- *Une convolution est une petit matrice (noyau) appliquée à une image.*
- *Les convolutions correspondent à des filtres généralement utilisés pour retrouver des templates particuliers dans une image.*
- *Le template matching utilise l'opérateur de corrélation croisée (cross-correlation), noté ⊗.*
- *Les opérateurs de convolution transforment l'image X en une nouvelle image Y de la façon suivante.*

$$Y_{i,j} = \sum_{l=-m}^{m} \sum_{c=-n}^{n} N_{l,c} X_{i+l,j+c}$$

*N représente le noyau de taille m*n utilisé comme filtre*

- *Concrètement, cette opération revient à faire glisser N sur l'image X, à multiplier les pixels qui se superposent et à sommer ces produits.*
- *Le template matching consiste à calculer la corrélation croisée entre X et le filtre N afin trouver des templates dans l'image X.*

CNN : Les features

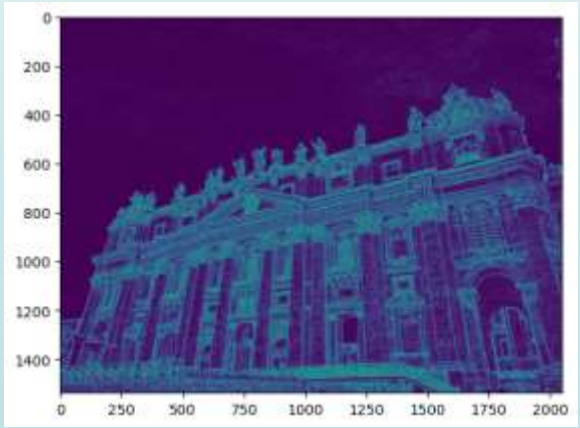
Les convolutions

- *Un problème de template matching se réalise en deux étapes : Détecter et décrire les features, (2) trouver les correspondances entre images.*
- *Les matrices de convolution sont généralement de taille 3*3 ou 5*5.*
- *Suivant les valeurs de coefficients elles peuvent être utilisées pour détecter les contours, augmenter le contraste, renforcer les bords ….*

$$\otimes \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = C_{i,j}$$

$$\otimes \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = C'_{i,j}$$

$$\sqrt{(C_{i,j})^2 + (C'_{i,j})^2} \Rightarrow$$



Application de deux produits de convolution à une image

CNN : Les features

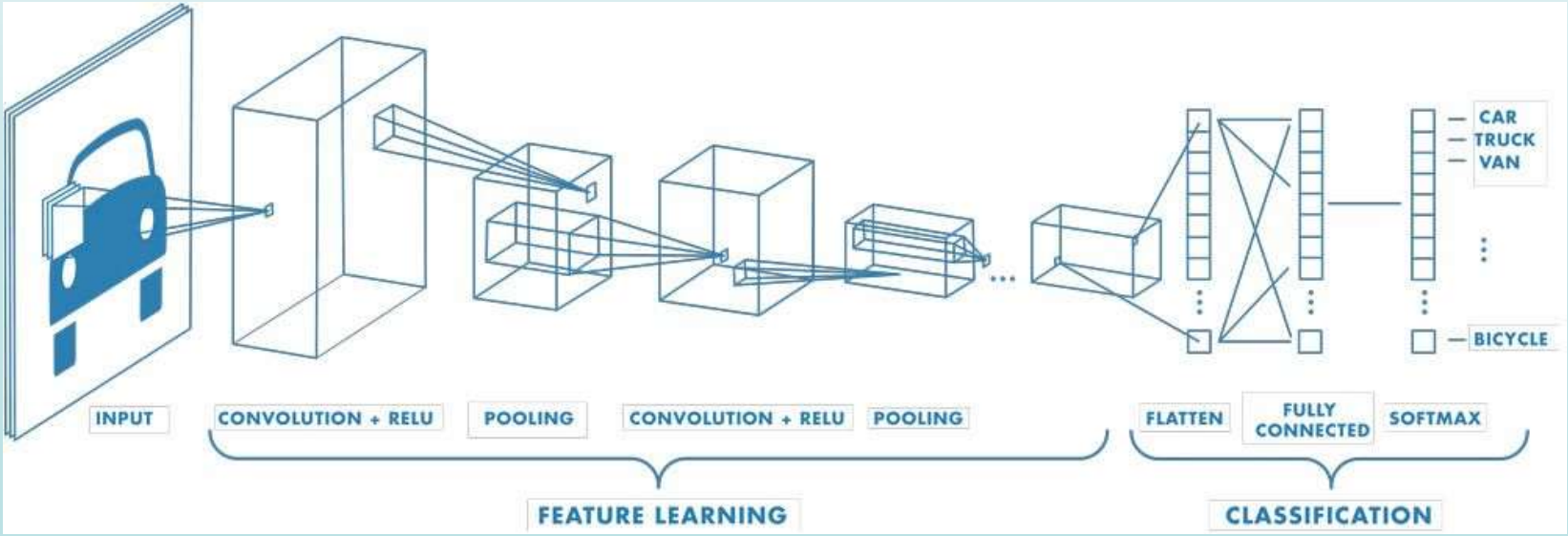
Les convolutions : padding et strides

- *Le noyau s'applique sur une partie de l'image sur le coin haut à gauche de l'image afin de produire un pixel pour l'image de sortie.*
- *Le noyau est ensuite décalé d'un nombre de cases horizontalement vers la droite pour obtenir un nouveau coefficient.*
- *Une fois en fin de ligne le noyau descend d'une case verticalement et repart de la droite, on parcourt ainsi toute la matrice image.*
- *Le nombre de déplacements correspond au stride horizontal et vertical.*
- *Si le stride est (1,1) l'image de sortie aura une taille m = n-f+1. ou n est la dimension de l'image et f celle du noyau.*
- *Dans le cas ou l'on souhaite que l'image de sortie ait la même taille que l'image initiale, il faut rajouter des zéros autour de la matrice initiale avant la convolution, on dit qu'on fait un padding.*
- *Si l'on joute p-1 contours de 0 à l'image initiale (p=f/2) et que le stride est de (1,1) alors l'image de sortie aura la même taille que l'image initiale.*

CNN : Les features

L'apprentissage des features

- *La vision par ordinateur consiste à extraire les features de chaque image du jeu de données, puis à entraîner un classifieur sur ces features*
- *Les performances des algorithmes d'apprentissage dépendent de la qualité des features trouvées.*
- *Mais avec les CNN et contrairement aux méthodes traditionnelles, (comme SIFT) les features sont apprises lors la phase d'entrainement.*



CNN : Les couches

La couche de convolution

- *La première couche de convolution (feature maps) détecte les features de l'image, comme les contours, les formes et les textures.*
- *Les couches suivantes détectent des features plus complexes à partir des features détectées par la couche précédente.*
- *Les filtres de chaque couche sont identiques (même poids). L'ensemble des neurones d'une couche détectent la même caractéristique dans différentes régions de l'image.*
- *Le réseau est alors entraîné à reconnaître un objet, quel que soit son emplacement dans l'image.*
- *La dernière couche de convolution est en général connectée à une couche de type MLP qui utilisent les features pour classifier les images.*
- *Après chaque opération de convolution, un CNN applique une transformation ReLU, afin d'introduire une non-linéarité dans le modèle.*
- *Pour diminuer la taille des images (donc le nombre de poids) les sorties des couches de convolution peuvent être redimensionnées (**pooling**).*

CNN : Les couches

La couche de pooling

- *Ces couches sont généralement placées entre deux couches de convolution.*
- *L'opération consiste à réduire la taille des images, tout en conservant les principales caractéristiques.*
- *L'image est découpée en cellules adjacentes de taille 2 × 2, ou bien de taille 3 × 3 qui se chevauchent sur une case. On conserve de la cellule que le pixel de valeur maximal.*
- *Le nombre de feature maps est conservé mais leurs tailles sont réduites de 3/4.*
- *Les couches de pooling permettent de réduire le nombre de poids, l'efficacité du réseau est amélioré et on évite le sur-apprentissage.*
- *En ne conservant que les pixels qui ont une valeur maximale on perd en précision. Le réseau est moins sensible à la position d'un point d'intérêt.*
- *C'est un avantage, car si la présence d'un point d'intérêt est nécessaire à la classification, la position ne doit pas rentrer en ligne de compte.*

CNN : Les couches

Architecture des CNN

- *Un CNN est un suite de couches de convolution (d'extraction de features), ReLU (activation), pooling (réduction) et de MLP.*
- *Les couches de convolution et de pooling reçoivent des matrices 3D (L,H,D), les MLP retournent toujours des vecteurs.*
- *L et H sont la largueur et la hauteur en pixels, D le nombre de canaux (1 pour les images en noir et blanc, 3 pour les images couleurs).*

Les paramètres des couches de convolution

- *Il est nécessaire de fournir au moins 4 hyperparamètres à une couche de convolution.*
- *F le taille des filtres carrée, le pas S avec lequel on fait glisser les filtres (droite et bas), P les épaisseurs du contour à ajouter de chaque coté.*
- *Une couche de convolution retourne une matrice de taille (Lc, Hc, D)*

$$L_c = \frac{L - F + 2P}{S} + 1 \qquad H_c = \frac{H - F + 2P}{S} + 1$$

CNN : Les couches

La couche fully-connected FCNNs

- *La couche fully-connected est toujours la dernière couche d'un réseau de neurones profond qu'il soit convolutif ou pas (LSTM, DBM, RBM …)*
- *Entièrement connecté signifie que chaque neurone de la couche cachée est connecté à chaque neurone de la couche d'entrée.*
- *Le fait d'être « entièrement connecté » (perceptron multicouche) permet aux informations de circuler librement entre tous les neurones du réseau.*
- *Le principal avantage de l'utilisation de ces couches entièrement connectées est qu'elles permettent aux réseaux profonds d'apprendre des relations complexes entre les data d'entrée et de sortie.*
- *Ces couches reçoivent un vecteur en entrée et applique une combinaison linéaire puis généralement une fonction d'activation.*
- *La sortie est souvent un vecteur, ou chaque élément du vecteur indique la probabilité pour l'image d'appartenir à une classe spécifique.*
- *Les FCNNs ont deux inconvénients majeurs qui sont une tendance à l'overfitting et l'explosion du nombre de paramètre à caler.*

CNN : Les couches

Les paramètres des couches de pooling

- *Deux paramètres sont nécessaires pour configurer une couche de pooling.*
- *La taille F des cellules carrées avec lesquelles ont découpe l'image, et S le nombre de pixels qui sépare chaque cellule.*
- *Une couche de pooling retourne une matrice de taille (Ls, Hs, D)*

$$L_s = \frac{L - F}{S} \qquad H_s = \frac{H - F}{S}$$

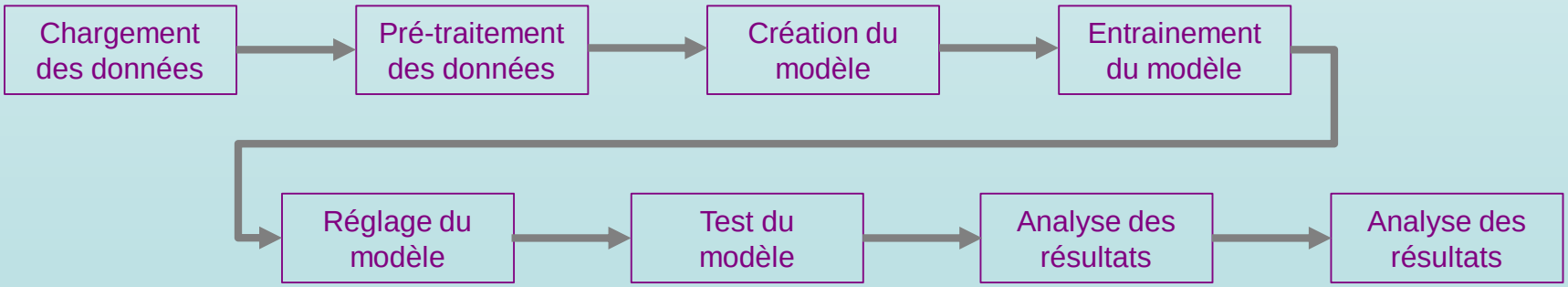
Les paramètres des couches fully-connected

- *Les images doivent être aplaties (couche Flatten) avant la couche MLP.*
- *Les principaux paramètres sont le nombre de couches, et de neurones.*
- *Pour chaque couche ont doit préciser les fonctions d'activation.*
- *Des couches de type Dropout peuvent être ajoutées après chaque couche de MLP ou de Convolution.*
- *Le taux de désactivation des neurones est le principal paramètre.*

CNN : Quick Draw

Reconnaissance de dessins

- *"Quick, Draw!" est une base de données composée de gribouillages (+50 Millions) correspondant à 340 catégories de dessins.*
- *Chaque image est une matrice carrée de 28 pixels nono-colore.*
- *On cherche à développer un CNN permettant de reconnaitre des images issues de ces bases de données.*
- *On se limitera à 10 catégories ('avion', 'fourmi', 'banane', 'bus', 'chat', 'chien', 'rivière', 'moustique', 'plage', 'chaise').*
- *L'application se décompose en 8 principales étapes.*

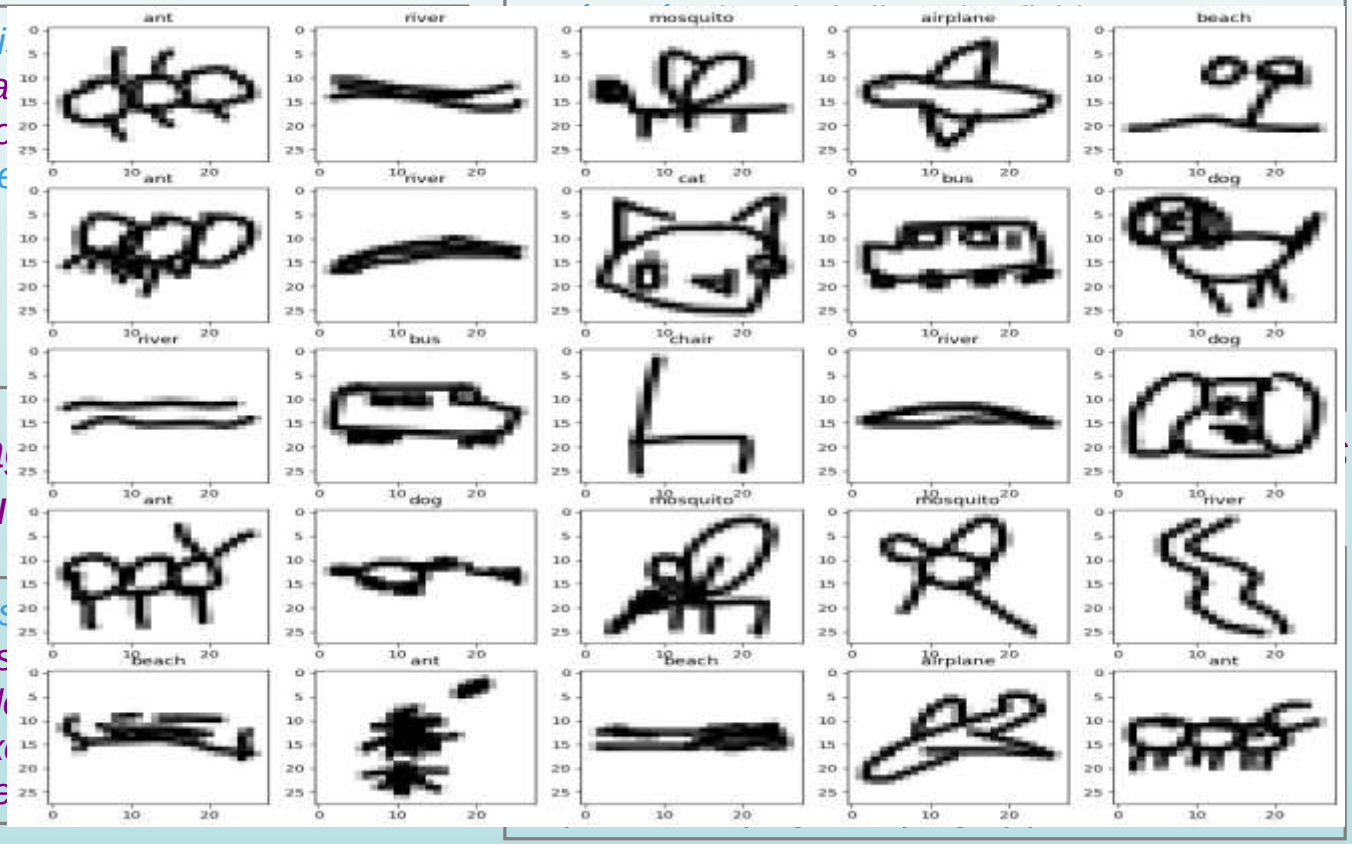


CNN : Quick Draw

Chargement des données

- *Les images sont téléchargées dans 10 fichiers depuis "Quick, Draw!".*
- *Seul une petite partie des images seront utilisées pour l'application.*

```
#Récupération de la li
directory = "Quick Dra
files = os.listdir(directo
#Création des variable
images = np.zeros((0,
targets = np.array([])
classes=[]
size_file = 2000
```



- *Les ima*

blancs, pu

```
#Mélange des indexes
indexes = np.arange(s
np.random.shuffle(ind
images=images[index
targets=targets[indexe
```

CNN : Quick Draw

Pré-traitement des données

- *Afin de faciliter l'algorithme d'apprentissage à mieux converger les images doivent être normalisées.*
- *Soit en utilisant un modèle de type StandardScalar; soit en divisant les images par la valeur 255.*
- *Les données doivent ensuite être découpées en une partie de test et une partie d'entrainement. Une découpe à 33% sera utilisée.*

Création du modèle

- *Il n'existe pas de démarche permettant de définir la configuration du réseau. Généralement cette démarche se fait plus par "tâtonnement ».*
- *La partie de feature-learning contiendra deux couches de convolution.*
- *La partie de classification de 3 couches.*
- *Une couche de mise à plat des images (flatten) doit être insérée entre ces deux parties, afin de transformer une image matrice (28,28,1) en un tableau (784,1).*

CNN : Quick Draw

Premier modèle

- *Le modèle est composé de 2 couches Conv2D avec 16 et 32 filtres, une couche Flatten et 3 couches Dense avec 256, 128 et 10 neurones.*
- *Les fonctions d'activation sont de type relu et softmax pour la dernière.*
- *Les matrices de convolution sont de taille 5*5 et 3*3.*

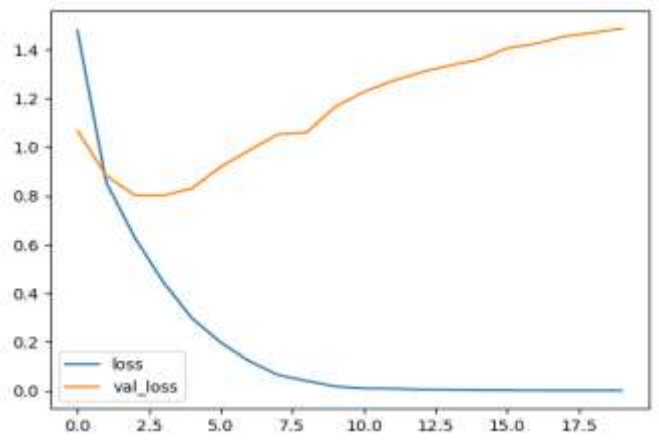
```
model.summary()        #Description du modèle
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 24, 24, 16)        416
conv2d_1 (Conv2D)            (None, 22, 22, 32)        4640
flatten (Flatten)            (None, 15488)             0
dense (Dense)                (None, 256)               3965184
dense_1 (Dense)              (None, 128)               32896
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 4004426 (15.28 MB)
Trainable params: 4004426 (15.28 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

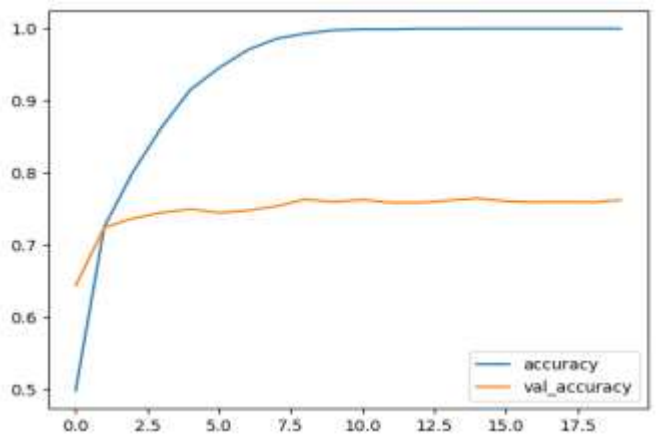
```
#Création d'un modèle séquentiel
model = tf.keras.models.Sequential()
#Partie de convolution
model.add(tf.keras.layers.Conv2D(16, 5, input_shape=(28,28,1), activation='relu'))
model.add(tf.keras.layers.Conv2D(32, 3, activation='relu'))
#Partie de convolution
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=256, activation='relu'))
model.add(tf.keras.layers.Dense(units=128, activation='relu'))
model.add(tf.keras.layers.Dense(units=10, activation='softmax'))
```

CNN : Quick Draw

Modèle d'apprentissage

- *On utilise le même modèle d'apprentissage que pour MNIST.*
- *Lors de l'apprentissage 20 epochs sont effectuées avec des tailles de batch de 256 et 20% des données pour la validation.*



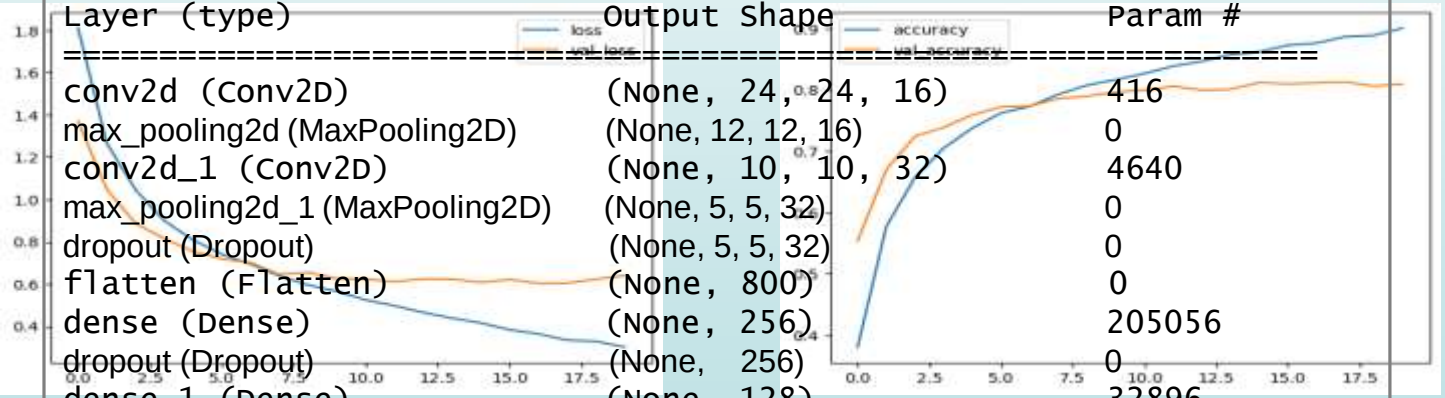


- *On constate que le modèle est très nettement en overfitting.*
- *Afin d'améliorer le modèle il est donc nécessaire de réduire la taille du réseau, en nombre de variables à caler.*
- *On peut supprimer des couches et/ou ajouter des couches pooling ou dropout.*

CNN : Quick Draw

Amélioration du modèle

- *On ajoute deux modèles de type MaxPooling2D avec des matrices de taille 2 × 2 après les couches Conv2D.*
- *Deux couches Dropout avec une désactivation de 20%, l'une après la partie de convolution et une après le premier modèle Dense.*
- *La taille du modèle est notablement réduite.*



Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 24, 24, 16)	416
max_pooling2d (MaxPooling2D)	(None, 12, 12, 16)	0
conv2d_1 (Conv2D)	(None, 10, 10, 32)	4640
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 32)	0
dropout (Dropout)	(None, 5, 5, 32)	0
flatten (Flatten)	(None, 800)	0
dense (Dense)	(None, 256)	205056
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 128)	32896
dense_2 (Dense)	(None, 10)	1290

Total params: 244298 (954.29 KB)
Trainable params: 244298 (954.29 KB)
Non-trainable params: 0 (0.00 Byte)

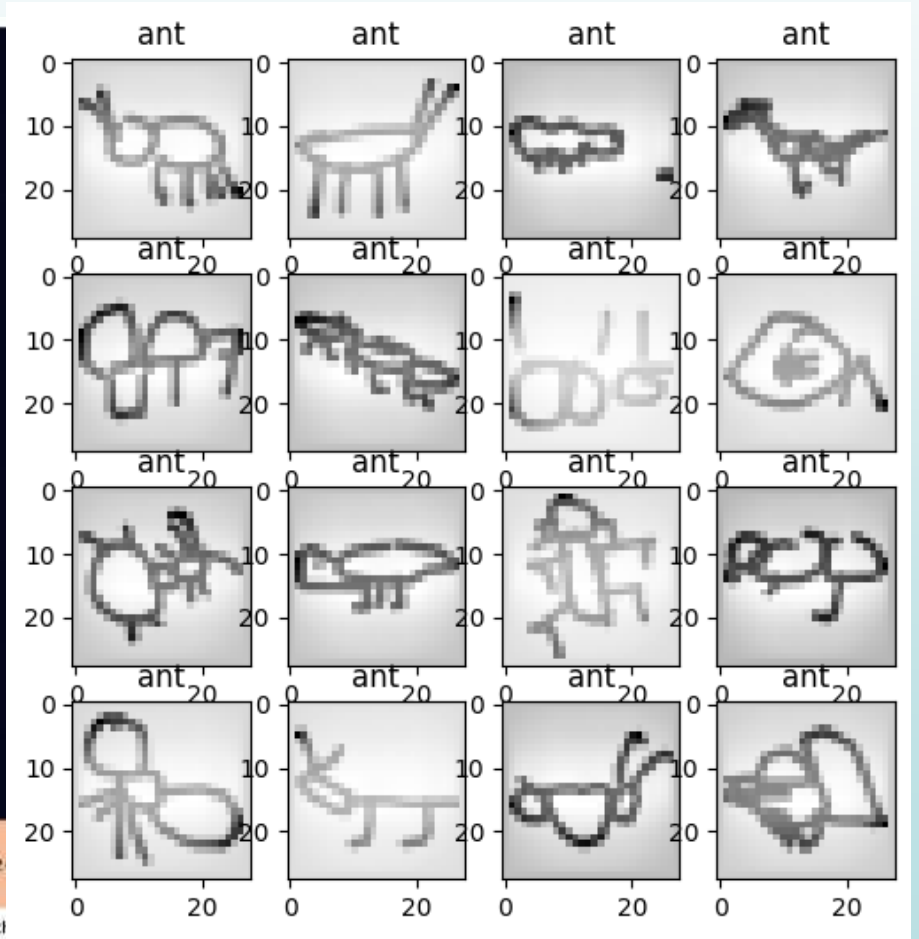
- *Le modèle n'est pas en situation de sur-apprentissage.*
- *On constate qu'à partir de 6 epochs le modèle n'apprend plus.*

CNN : Quick Draw

Résultat final

Le choix de désactiver et de réduire la taille des images permet d'améliorer les résultats.

Affichage des fournis classées en chien



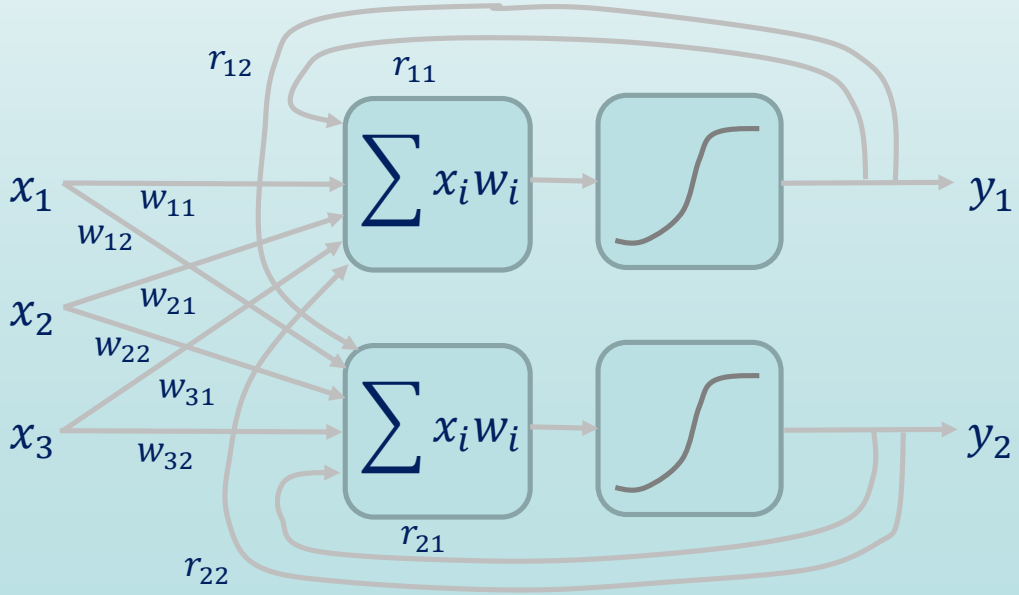
Long Short-Term Memory : LSTM

- Fonctionnement
- Prédiction financière

Long Short-Term Memory : LSTM

Les réseaux récurrents : RNN

- *Les réseaux de neurones tels que les MLP ou les CNN ne sont pas en mesure de faire des prédictions sur des séries temporelles.*
- *Les RNN eux sont des réseaux adaptés au traitement des séquences.*
- *Afin de prendre en compte la sérialité des données les sorties obtenues à l'itération t-1 sont utilisées pour calculer les sorties à l'itération t.*



$$Y^t = \sigma(WX + RY^{t-1} + B)$$

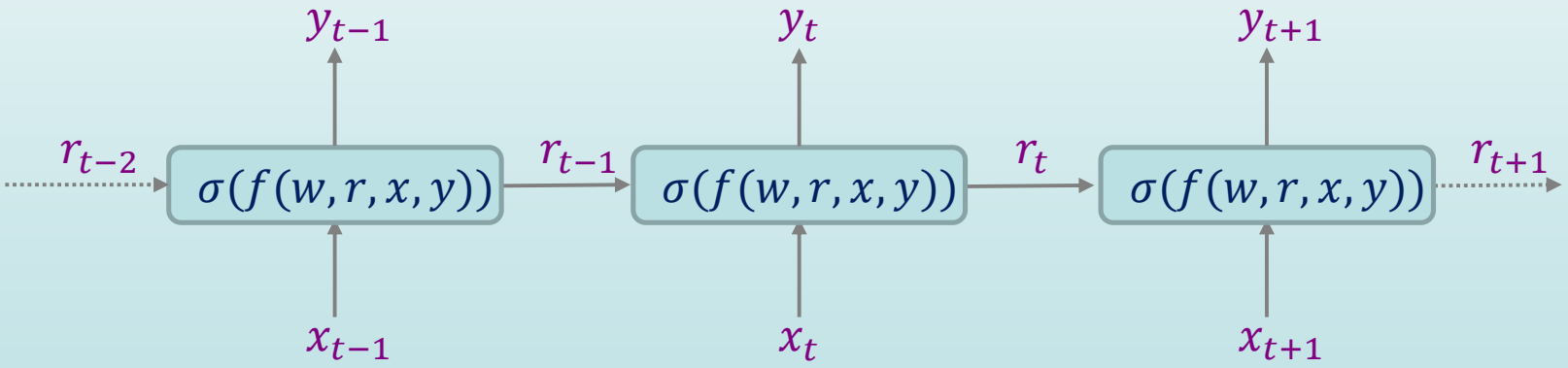
Ou les r_{ij} sont les poids de la récurrence

Les neurones récurrents sont utilisés ensemble sur une couche (appelé cellule)

LSTM : Fonctionnement

Dépilement des RNN

- *L'utilisation des RNN permet de modéliser les dépendances qui peuvent exister entre les éléments de la séquence.*
- *La forme classique ne fait pas apparaître la dimension temporelle.*
- *On utilise alors une représentation du réseau "déplié dans le temps", afin de faire apparaître explicitement la notion de temps.*

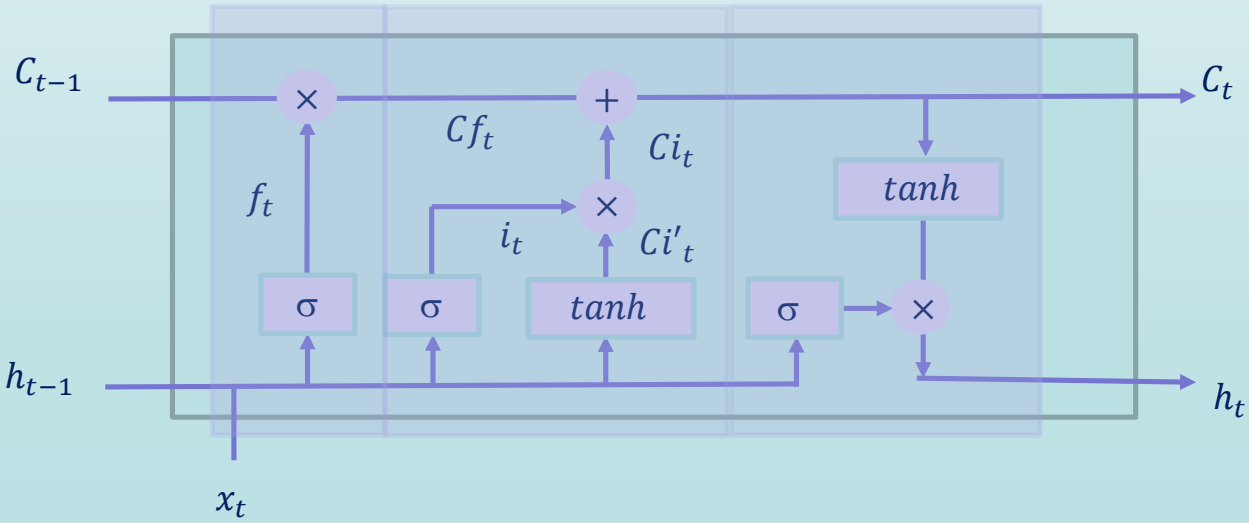


- *Un RNN peut être utilisé pour prédire une seul sortie y_n pour une série d'entrée $x_1 \ldots x_n$ "many to one".*
- *On peut également utiliser les RNN pour des prédictions de type "one to many" ou "many to many".*

LSTM : Fonctionnement

Long Short Term Memory

- *Les RNN ne peuvent pas modéliser les dépendances sur le long terme.*
- *Les LSTM possèdent une mémoire interne qui leurs permet de conserver un état aussi longtemps que nécessaire.*
- *Avant chaque fonction d'activation on dispose d'une couche de neurones (ou cellules) composée de N_h + N_i neurones.*
- *La taille des Ct est égale à N_h.*



Forget Gate

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

$$C_t = C_{t-1} \times f_t$$

Input Gate

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

$$Ci'_t = tanh(W_c[h_{t-1}, x_t] + b_c)$$

$$Ci_t = Ci'_t \times i_t \quad \text{et.} \quad Ci_t = Ci_t + Ci_t$$

Ouput Gate

$$O_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

$$h_t = \tanh(C_t) \times O_t$$

LSTM : Fonctionnement

Données d'apprentissage

- *Afin de réaliser l'apprentissage des cellules; des LSTM sont entrainées à partir de matrices de dimension trois.*
- *On doit définit le nombre d'échantillons ou de séquences (NS), c'est-à-dire le nombre de batch qui servirons à entrainer et à tester le modèle.*
- *La taille des séquence (TS). Chaque séquence est une suite de variables ($X_0..X_n$) qui doivent être traiter simultanément par le modèle.*
- *Le traitement d'un séquence permet d'obtenir une sortie Y_n après son traitement complet par le réseau pour un apprentissage "many to one".*
- *Dans les autres cas plusieurs sorties Y peuvent être obtenues.*
- *Enfin la taille (TV) des variables d'entrée X_i, doit également être fourni.*
- *Une couche LSTM prend en entrée une séquence de la matrice d'origine (matrice de taille TS x TV).*
- *La couche retourne un vecteur Y de taille égale au nombre de neurones de la couche.*

LSTM : Prédiction

Prédiction financière

- *Le modèle consiste a prédire l'évolution du cours en bourse d'une société (Texas-Instrument).*
- *Les données sont téléchargées depuis "yahoo finance", sur une période 5 ans du 4 mars 2019 au 1 mars 2024.*
- *L'objectif est d'entrainer le modèle sur un paramètre (valeur de l'action à la clôture) sur 4 années et de prédire son évolution sur la dernière.*

LSTM : Prédiction

Préparation des données

- *Les LSTM doivent traiter les données dans l'ordre chronologique.*
- *Le cours de l'action à la date t doit obligatoirement être traité avant le cours de l'action à la date t+1.*
- *Les données sont fournies au réseau sous forme de séquences.*
- *Une séquence est un suite de cotations aux dates [i,i+1,…i+TS] et la target à estimer est la cotation à la date i+TS+1.*
- *La même feature sert pour les valeurs d'entrée (X) et pour la target.*

Date	Close	Close
2019-03-04	107.400002	107.400002
2019-03-05	106.570000	106.570000
2019-03-06	105.419998	105.419998
2019-03-07	104.949997	104.949997
2019-03-08	104.860001	104.860001
2019-03-11	106.449997	106.449997

Taille des séquences TS=4

La première séquence est composée des cotations du 4 au 7 mars 2019, elles servent à prédire la cotation le 8

*Avec un **stride** de 1 la seconde séquence est décalée de 1 afin que les 4 valeurs du 5 au 8 servent à l'estimation du 9.*

LSTM : Prédiction

Préparation des données

- Comme les algorithmes de retro-propagation sont sensibles aux valeurs, il est conseillé de standardiser les données (StandardScaler).
- Les données sont organisées en séquences composées de cotations consécutives et d'une valeur à prédire qui est la cotation suivante.
- TimeseriesGenerator de keras permet de créer ce type de séquences.

```
#Standardisation des données
scalar = StandardScaler()
X = scalar.fit_transform(np.array(data['Close']).reshape(-1,1))
#Découpage des données en train et en test 80/20
from keras.preprocessing.sequence import TimeseriesGenerator
X_tr, X_te = X[:int(len(X)*0.8)], X[int(len(X)*0.8):]
X_train = TimeseriesGenerator(X_tr, X_tr, length=10, stride=1, batch_size=1)
X_test = TimeseriesGenerator(X_te, X_te, length=10, stride=1, batch_size=1)
```

```
X_train[0]
(array([[
[-1.8143269], [-1.84506518],
[-1.8876543], [-1.90506029],
[-1.9083932], [-1.84950937],
[-1.8469169], [-1.80766082],
[-1.8254372], [-1.69063369]]
]),
array([[-1.72507526]]))
```

- Les séries sont de taille TS=10 (length), pour estimer l'estimation d'une cotation à TS+1.
- D'une série à l'autre les valeurs sont "switché" d'une cotation (stride=1) et les mises à jour des poids se font à chaque étape (batch_size=1)

LSTM : Prédiction

Modèle récurrent

- *Le modèle est composé de deux couches de type LSTM de 50 neurones chacune.*
- *Les fonctions d'activation sont des 'relu' et la dimension des séquences d'entrée sont de taille (10,1).*
- *10 est la longueur des séquences et 1 le nombre de features.*
- *Une couche Dense avec 1 neurone pour les sorties est ajoutée.*

```
#Création d'un modèle LSTM
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.LSTM(50, activation='relu', return_sequences=True, input_shape=(10,1)))
model.add(tf.keras.layers.LSTM(50, activation='relu'))
model.add(tf.keras.layers.Dense(1))
```

```
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 10, 50)            10400
lstm_1 (LSTM)                (None, 50)                20200
dense (Dense)                (None, 1)                 51
=================================================================
Total params: 30651 (119.73 KB)
Trainable params: 30651 (119.73 KB)
Non-trainable params: 0 (0.00 Byte)
```

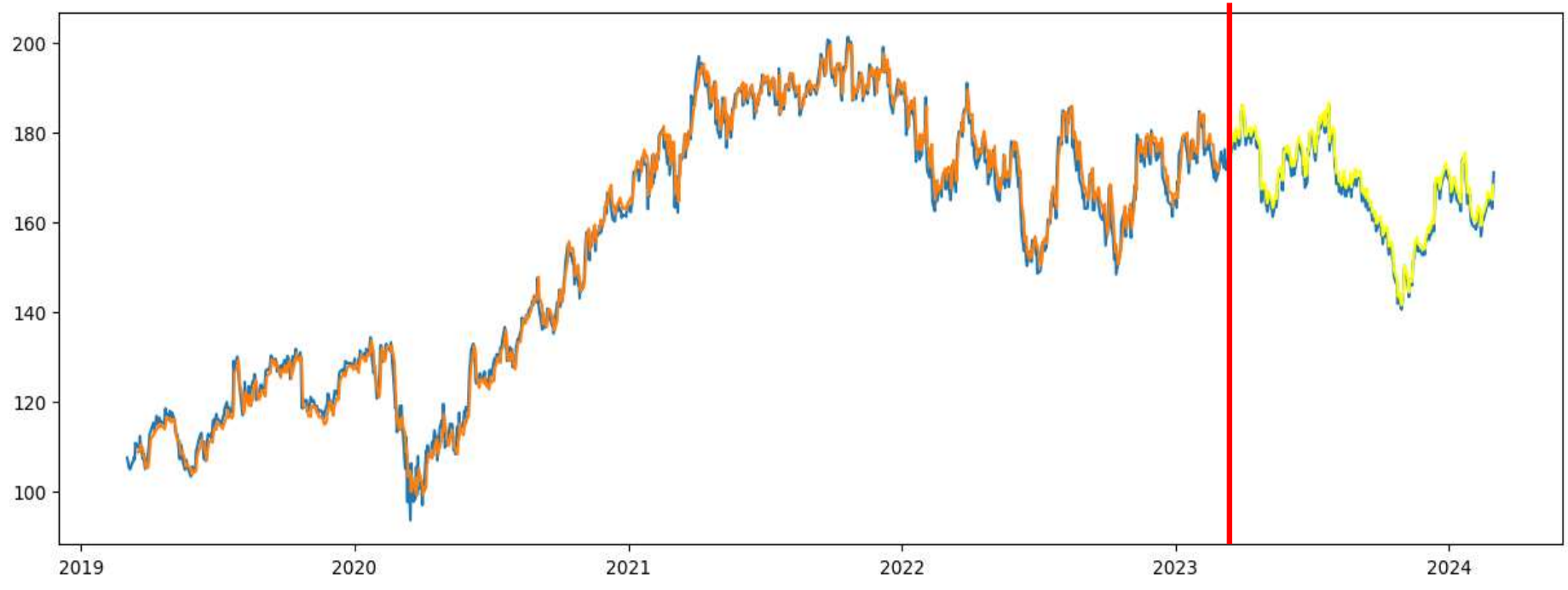
- *La première couche LSTM a une mémoire de 50 (N_h), elle est composée de 4 "gates" identiques, et les X sont de dimension 1 (N_i).*

$$10400 = 4 \times ([N_h + N_i] \times N_h + N_h)$$

LSTM : Prédiction

Apprentissage

- *Pour la méthode d'apprentissage l'optimiser est la descente de gradient (méthode Adam) et l'erreur quadratique est la fonction à minimiser.*
- *L'apprentissage se fait directement sur le X_train, avec un nombre d' époch de 10.*



LSTM : NLP

Les couches LSTM pour traitement du langage

- *Afin de pouvoir analyser des textes, il est nécessaire de mettre en forme les séquences (tokenisation, padding et embedding).*
- *Les mots doivent être numérotés : tokensization ;*
- *Les séquences doivent avoir toutes la même taille (padding)*
- *Chaque mots doit être représenté par un vecteur qui puisse donner du sens aux mots en fonction de leur utilisation (embedding)*

Tokenization et padding

- *Plusieurs outils de tokenisation existent, en particulier dans Keras.*

```
#Création d'une liste de liste à partir des séquences dans un dataframe['text']
sequences = [x.split() for x in data['text'].tolist()]
```

```
#Tokenisation des séquences + découpe des séquences à une taille fixe : sequences_size
tokenizer = keras_preprocessing.text.Tokenizer()          #création d'une tokeniser
tokenizer.fit_on_texts(sequences)                         #Apprentissage
sequences=tokenizer.texts_to_sequences(sequences)         #Tokenisation des séquences
sequences=pad_sequences(sequences,sequences_size) .
```

LSTM : NLP

L'embedding des mots : Word2Vec

- L'embedding peut se faire avant l'apprentissage ou par le modèle.
- Word to Vector est un algorithme d'apprentissage développé par Google pour transformer des mots en vecteurs de sens.
- Ce modèle propose deux architectures principales d'apprentissage.
- CBOW calcul l'embedding des mots à partir d'une fenêtre de mots.
- Skip-gram : prédit les mots entourant (fenêtre) un mot central.

```
#Modèle Word2Vec de la bibliothèque gensim avec des vecteurs de taille : embedding_size
#taille des fenêtres d'apprentissage : 10 mots. Prise en compte des mots courants mint_count=1
w2v=gensim.models.Word2Vec(sentences= sequences, vector_size=embedding_size, window=10,
min_count=1)
#Création d'une matrice d'embedding pour le pré-apprentissage
vocab_size=len(tokenizer.word_index)+1        #Un mot de plus que la tokenisation (pour mots vide)
weight_of_word = np.zeros((vocab_size, embedding_size))
#Création d'une table de correspondance entre les numéros des tokens et leur emdedding
for word, index in tokenizer.word_index.items() :     #Parcours de tous les tokens créés
        weight_of_word[index]=w2v.wv[word]
```

LSTM : NLP

Modèle Bidirectionnel

- *Les couches récurrentes LSTM ou GRU traitent les séquences dans une seule direction, **c'est**-à-dire sans le sens "fifo".*
- *Les couches bidirectionnelles permettent de traiter les séquences dans les deux sens. Elle sont particulièrement utilisée pour les tâches de traitement du langage naturel.*
- *Les couches bidirectionnels permettent mieux capturer les dépendances à long terme dans les séquences.*

```
#Création d'un modèle Bidirectionnel LSTM pour la classification de textes
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Input(shape=(seq_size,)))   #seq_size : taille des séquences
#Vocab_size : taille du vocabulaire ; embedding_size : taille des vecteurs d'embedding ;
#embedding_dict : dictionnaire d'embedding (lien entre les tokens et leur vecteur d'embedding)
model.add(keras.layers.Embedding(input_dim=vocab_size, output_dim=embedding_size,
                                 weights=[embedding_dict], trainable=False))
model.add(tf.keras.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True))
model.add(tf.keras.Bidirectional(tf.keras.layers.LSTM(32))
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(1, activation='softmax'))
```

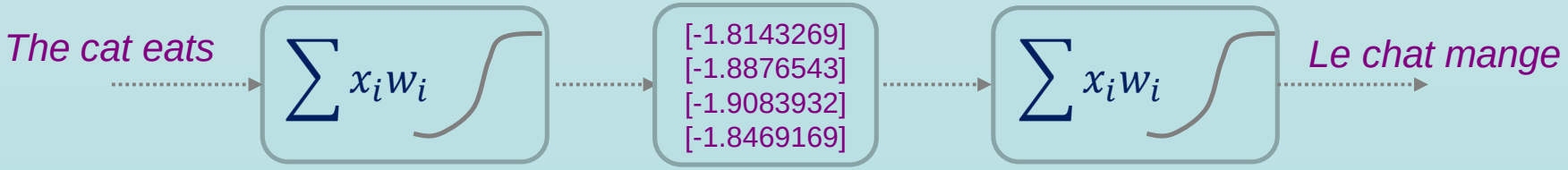
Encodeur Décodeur

- Description
- Les auto-encodeurs
- Modèle de suppression du bruits

Encodeur-Décodeur

Les encodeurs et décodeurs

- *Les Encodeur-Décodeurs sont des réseaux de neurones découvert en 2014. C'est un des piliers du Outils et Méthodes de l'IA.*
- *Ils sont utilisés pour des tâches telles la traduction, mais également des tâches de compression ou de détection de motifs.*
- *Un encodeur-décodeur est un modèle de Outils et Méthodes de l'IA composé de deux réseaux de neurones.*
- *Le premier réseau de neurones prend une phrase en entrée pour en sortir une séquence de chiffres.*
- *Le second réseau récupère cette séquence pour en sortir une phrase.*
- *Ces deux réseaux font la même chose, simplement l'un est utilisé dans la direction normal et l'autre et dans le sens opposé.*



Encodeur-Décodeur

Les avantages d'un encodeur-décodeur

- *Si l'on traduit une phrase avec une approche classique, la traduction se fait mot a mot sans prendre en compte le sens global de la phrase.*
- *L'objectif d'un encodeur est d'extraire le sens de la phrase et les relations des mots en fonction du contexte.*
- *Le décodeur peut analyser le vecteur pour produire sa propre phrase, qui ne dépend plus uniquement de l'ordre des mots.*

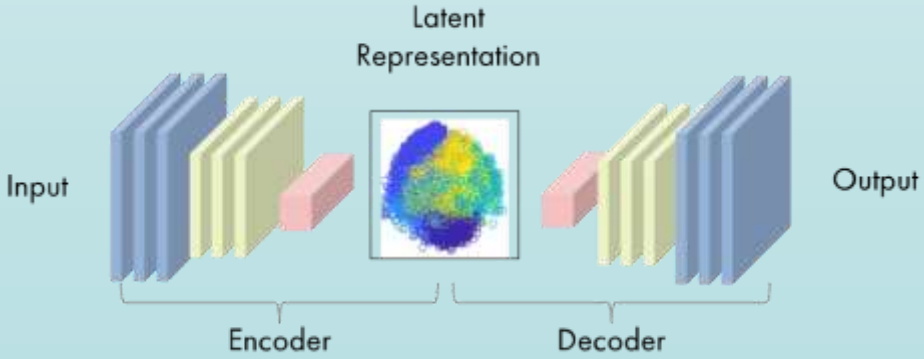
Les inconvénient d'un encodeur-décodeur

- *Les résultats produits par les encodeurs sont fixes.*
- *Ils sont efficaces pour traiter des phrases petites mais dès que la phrase est trop longue, le vecteur ne pourra pas stocker l'information nécessaire.*
- *Les Encodeur-Décodeurs doivent être couplés à des mécanismes qui leur permettent de s'adapter aux phrases de toutes tailles.*
- *Ils doivent être couplés à un mécanisme spécifique l'**attention**.*

Encodeur-Décodeur

Les Auto-Encodeurs AE

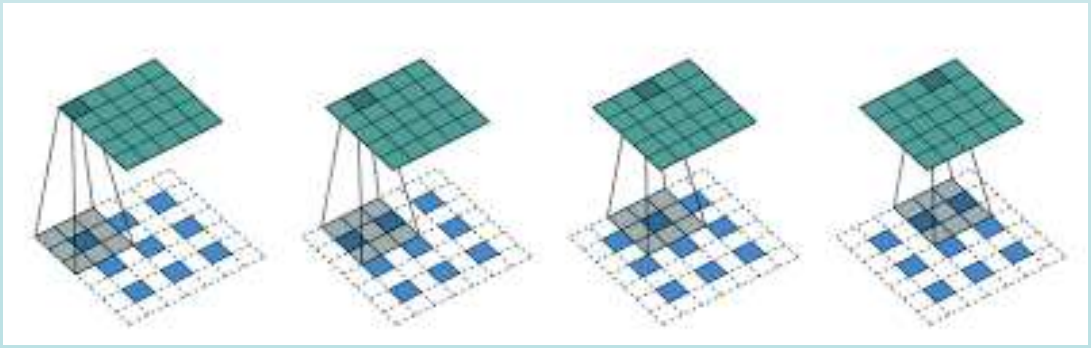
- *Les AE sont des réseaux qui possèdent le même nombre de neurones sur leur couche d'entrée et leur couche de sortie.*
- *L'encodeur sert à compresser les données d'entrée en une représentation plus petite.*
- *Il en extrait les caractéristiques (features) les plus importantes. Cette réduction se traduit par la suppression du bruit afin de ne laisser passer que les informations importantes.*
- *Un vecteur compacte appelé espace latent (ou bottleneck) est obtenu.*
- *A partir des caractéristiques contenues dans le vecteur condensé le décodeur doit reconstruire le plus fidèlement possible le jeu de données.*



Encodeur-Décodeur

Modèle de suppression du bruits d'images MNIST

- *L'encodeur est composé de couches de convolutions qui vont réduire des images (28,28) en vecteurs de petites tailles.*
- *Cette première couche ne doit retenir que les caractéristiques importantes des images, et omettre les informations non pertinentes.*
- *La couche de décodage est composée de couches de convolution particulières : des couches de convolution transposées.*
- *Ces couches sont l'exacte opposée des couches de convolution, elles doivent rééchantillonner les images à partir de la représentation latente.*
- *Deux paramètres définissent ces couches : strides, padding*



Le stride est l'écartement entre les pixels de limage de départ. Le nombre de 0 ajouté entre les pixels est égal à s-1.
Ici 1 zéro donc s=2

Le padding correspond à la taille des contours ajoutés. Ce nombre es<t k-p-1
Ici k=3 donc p=1.

*La taille finale est (n-1)*s + k – 2*p.*
*Soit (3-1)*2 + 3 - 2*1 = 5*

Encodeur-Décodeur

Structure du réseau

- *Les couches de convolution de l'encodeur auront la charge d'extraire les caractéristiques importantes de l'image et de réduire leurs tailles.*
- *La taille des images peut être réduite soit grâce aux variables strides et padding, soit en ajoutant des couches de pooling après les convolutions.*
- *Les images obtenues sont traitées au sein de plusieurs filtres.*
- *Le vecteur compacte peut être obtenu à partir de couches Denses.*
- *La structure du décodeur est l'exacte inverse de celle de l'encodeur.*

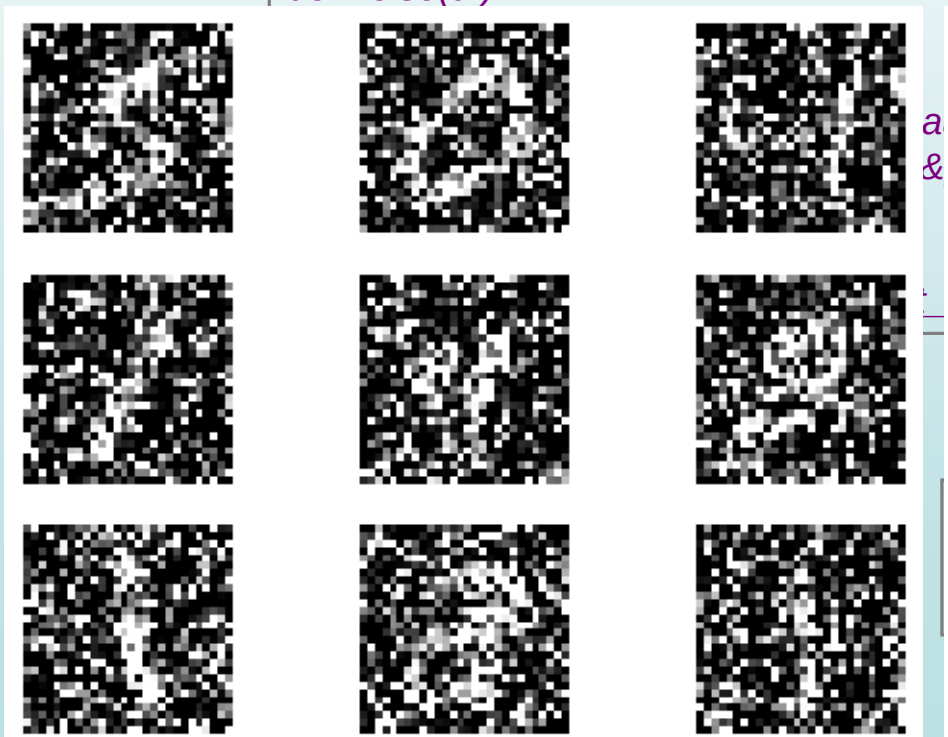
```
#Chargement de images
(X_train,Y_train),(X_test,Y_test) = mnist.load_data()
#Réduction du nombre de digits
X_train=X_train[:20000]; Y_train=Y_train[:20000]
X_test=X_test[:2000] ; Y_test=Y_test[:2000]
#Normalisation des données
X_train = np.reshape(X_train, [-1, input_size])
X_train = X_train.astype('float32')/256
X_test = np.reshape(X_test, [-1, input_size])
X_test = X_test.astype('float32')/256
```

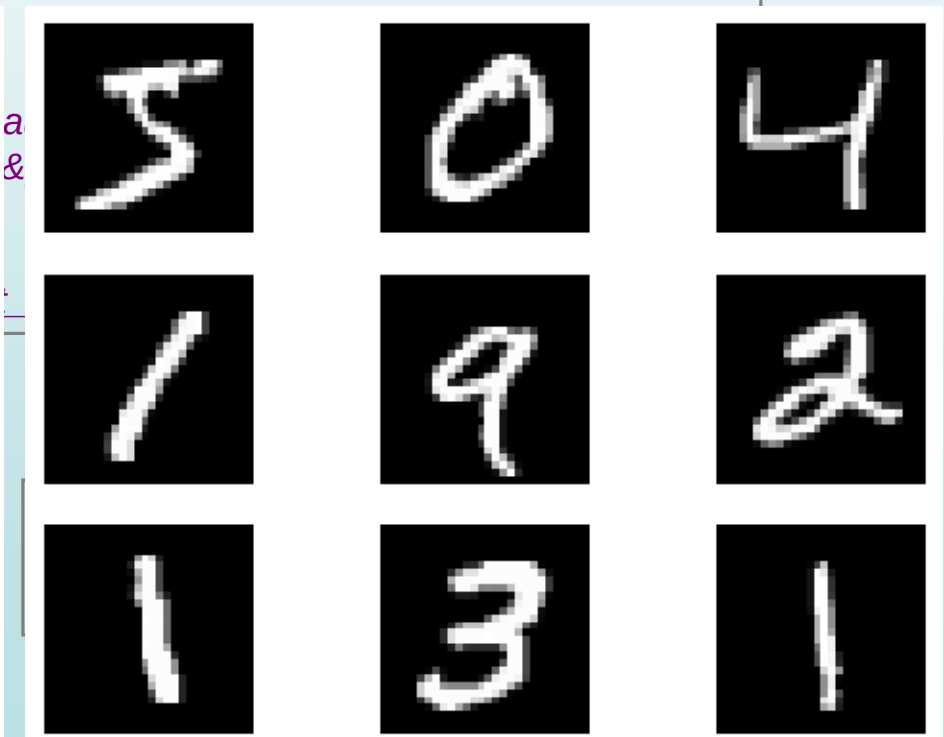
Encodeur-Décodeur

Préparation des données

- *La fonction random_noise de skimage.util permet d'ajouter différents sorte de bruits sur des images.*

```
#Création d'images bruitées
def noise(df):
```





Encodeur-Décodeur

Création du modèle

- *L'encodeur est composé de deux couches de convolution, une couche flatten pour aplatir les images et de deux couches denses pour obtenir un vecteur lattent.*
- *Le décodeur réalise le processus inverse.*

```
#Encodeur
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Conv2D(32, 3, input_shape=(28,28,1), activation='relu', strides=2, padding='same'))
model.add(tf.keras.layers.Conv2D(64, 3, activation='relu', strides=2, padding='same'))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=16, activation='relu'))
model.add(tf.keras.layers.Dense(units=10))
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 14, 14, 32)	320
conv2d_1 (Conv2D)	(None, 7, 7, 64)	18496
flatten (Flatten)	(None, 3136)	0
dense (Dense)	(None, 16)	50192
dense_1 (Dense)	(None, 10)	170
dense_2 (Dense)	(None, 3136)	34496
reshape (Reshape)	(None, 7, 7, 64)	0
conv2d_transpose (Conv2DTranspose)	(None, 14, 14, 64)	36928
conv2d_transpose_1 (Conv2DTranspose)	(None, 28, 28, 32)	18464
conv2d_transpose_2 (Conv2DTranspose)	(None, 28, 28, 1)	289

Total params: 159355 (622.48 KB)
Trainable params: 159355 (622.48 KB)
Non-trainable params: 0 (0.00 Byte)

```
#Décodeur
model.add(tf.keras.layers.Dense(units=7*7*64, activation='relu'))
model.add(tf.keras.layers.Reshape((7,7,64)))
model.add(tf.keras.layers.Conv2DTranspose(64, 3, activation='relu', strides=2, padding='same'))
model.add(tf.keras.layers.Conv2DTranspose(32, 3, activation='relu', strides=2, padding='same'))
model.add(tf.keras.layers.Conv2DTranspose(1, 3, activation='sigmoid', padding='same'))
```

Encodeur-Décodeur

Modèle d'apprentissage

- *La méthode d'apprentissage l'optimiser est une descente de gradient (méthode Adam).*
- *La binary_crossentropy est une fonction de perte adaptée à la compara*