

Université Alioune Diop de Bambey

Département TIC

Deep Learning

Convolutional Neuronal Network Recurrent Neuronal Network

Dr Seydou Nourou Sylla

seydounourou.sylla@uadb.edu.sn, nourou03@gmail.com

July 5, 2023



Ressources

Classification Supervisée

Définitions

Historique

Modèles de Perceptrons

Fonctions d'activation

Fonctions d'activation

Perceptron

Cost Function

Back Propagation

Convolutional Neuronal Network

Traitement d'image

Définition

Paramètres

Problématique

Compulation

Fonction Loss

Application

Preprocessing the data



- ▶ UCI Machine Learning Repository
- ▶ MNIST Database of handwritten digits(<http://yann.lecun.com/exdb/mnist/>)
- ▶ Keras (<https://keras.io/>)
- ▶ ImageNet (<https://www.image-net.org/>)
- ▶ Face Recognition Database(<https://www.face-rec.org/databases/>)
- ▶ Public Data Sets on AWS
- ▶ Kaggle



Pour maîtriser l'apprentissage supervisé, il faut absolument comprendre et connaître les 4 notions suivantes :

1. Les Données
2. Le Modèle et ses paramètres
3. La Fonction Coût
4. L'Algorithme d'apprentissage



On considère n objets (images, textes, son. . .), décrits par p caractéristiques.

Pour chaque objet i , ces caractéristiques sont stockées dans un vecteur

$$x_i = (x_1, \dots, x_p)$$

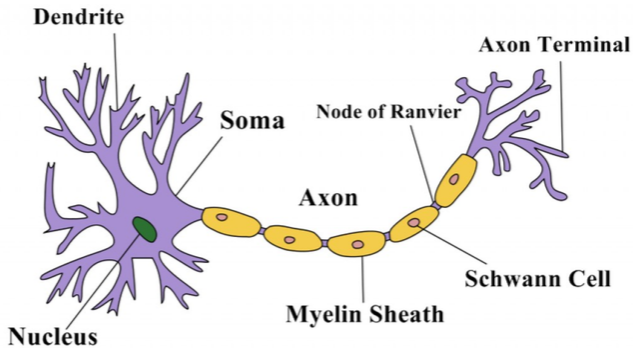
A chaque objet i est affectée une variable de sortie y_i .

- ▶ Si $y_i \in \mathbf{R}^p$ on parle de régression
- ▶ Si $y_i \in E$ avec E ensemble fini ($E = \{0, 1\}; E = \{0, \dots, 9\}$)), on parle de discrimination, classement, reconnaissance de forme

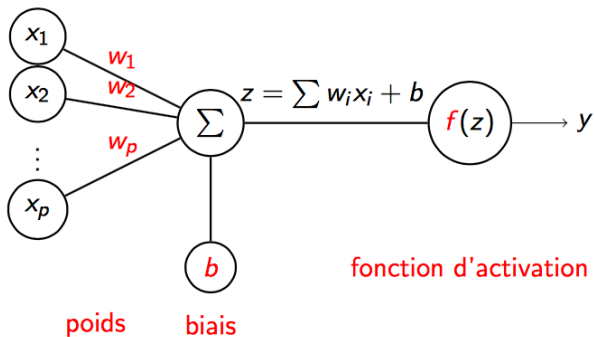
But: prédire la sortie y pour un nouvel ensemble de caractéristiques X

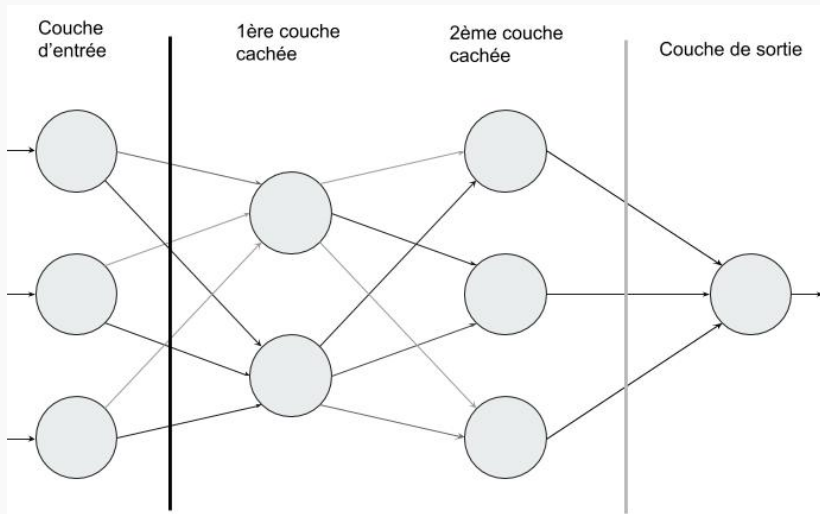
Comment: apprendre (sur un jeu de données d'apprentissage = training) une règle de prédiction ou classification et fournir cette règle pour l'appliquer à X

Neurone



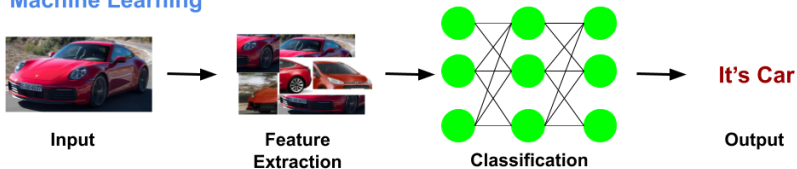
Un neurone formel



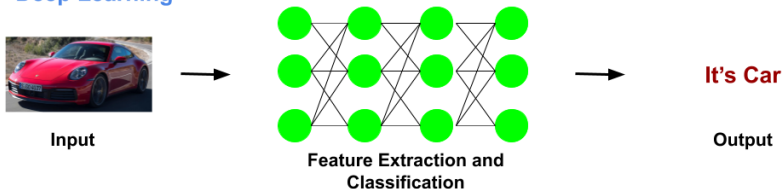




Machine Learning



Deep Learning





Définition 1

Les réseaux neuronaux artificiels sont des modèles mathématiques inspirés du cerveau humain, en particulier de sa capacité à apprendre, à traiter et à exécuter des tâches.

Définition 2

Les réseaux de neurones artificiels sont des outils puissants qui aident à résoudre des problèmes complexes liés principalement au domaine de l'optimisation combinatoire et de l'apprentissage automatique.

Définition 3

Technique de Machine Learning qui permet aux machines de d'apprendre des patterns ou des représentations complexes dans les données.



Nous pouvons dire que les réseaux neuronaux artificiels sont des méthodes puissantes qui peuvent donner aux ordinateurs une nouvelle possibilité, c'est-à-dire qu'une machine ne reste pas bloquée sur des règles préprogrammées et s'ouvre à diverses options pour apprendre de ses propres erreurs.



- 1943: Neural networks
- 1957: Perceptron
- 1974-86: Backpropagation, RBM, RNN
- 1989-98: CNN, MNIST, LSTM, Bidirectional RNN
- 2006: “Deep Learning”, DBN
- 2009: ImageNet
- 2012: AlexNet, Dropout
- 2014: GANs
- 2014: DeepFace
- 2016: AlphaGo
- 2017: AlphaZero, Capsule Networks
- 2018: BERT



- Mark 1 Perceptron – 1960
- Torch – 2002
- CUDA – 2007
- Theano – 2008
- Caffe – 2014
- DistBelief – 2011
- TensorFlow 0.1 – 2015
- PyTorch 0.1 – 2017
- TensorFlow 1.0 – 2017
- PyTorch 1.0 – 2017
- TensorFlow 2.0 – 2019



- ▶ Modele

Nous n'avons plus un modèle simple du type

$$Y = f(x) = ax + b$$

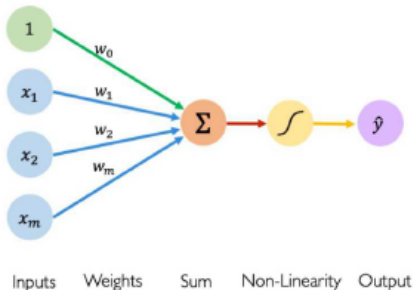
Mais plutôt

un réseau de fonctions connecté les unes aux autres

Ce qui correspond a un reseau de neurones.

- ▶ Algorithme d'optimisation
- ▶ Minimisation des Erreurs

Le Perceptron est le premier modèle en réseaux de neurones. Il permet de classifier de manière linéaire 2 classes.



$$\hat{y} = g(w_0 + \mathbf{X}^T \mathbf{W})$$

$$\text{where: } \mathbf{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \text{ and } \mathbf{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$



- ▶ **(Inputs):** x_1, x_2, \dots, x_n sont les caractéristiques des données
- ▶ **(Weights):** w_1, w_2, \dots, w_n sont les coefficients associés à chaque entrée.
- ▶ **(Biases):** est comme l'interception ajoutée dans une équation linéaire. C'est un paramètre supplémentaire dans le réseau neuronal qui est utilisé pour ajuster la sortie ainsi que la somme pondérée des entrées du neurone.
- ▶ **(Sum \sum):** est une fonction qui additionne les produits entre les entrées x_i et les poids w_i

$$z = x_1 \times w_1 + x_2 \times w_2 + \dots + x_m \times w_m + b$$

- ▶ **(Fonction Activation g):** C'est une fonction non-linéaire qui permet de gérer les contraintes non-linéaire sur les données
- ▶ **(Output):** C'est la sortie du réseau. C'est le résultat des calculs de $\hat{y} = g(w_o + X^T W)$



Exercice 1

Écrire une fonction qui renvoie z

Fonctions d'activation

Fonctions d'activation



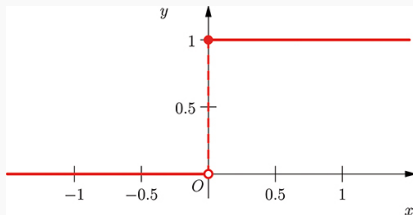
Une fonction d'activation n'est rien d'autre qu'une fonction $f : \mathbb{R} \rightarrow \mathbb{R}$ qui sera dans la suite associée à un neurone.

Nous allons en étudier plusieurs exemples. À une entrée $x \in \mathbb{R}$, on associe une sortie $y = f(x)$.



C'est la fonction marche d'escalier définie par la formule suivante

$$g(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$$



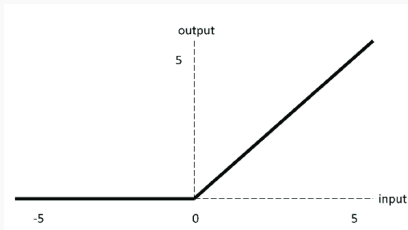


ReLU est une fonction linéaire par morceaux(elle est linéaire à gauche et linéaire à droite). Une fonction définie par

$$g(z) = \begin{cases} a \times z & \text{si } z < 0 \\ b \times z & \text{si } z \geq 0 \end{cases}$$

La plus utilisée est la fonction ReLU (pour Rectified Linear Unit), définie par

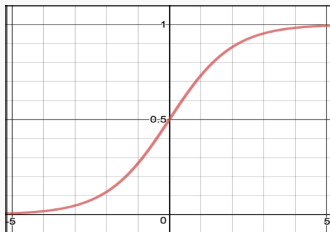
$$g(x) = \begin{cases} 0 & \text{si } z < 0 \\ z & \text{si } z \geq 0 \end{cases}$$





La fonction sigmoïde est définie par :

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



La fonction ressemble un peu à la fonction de Heaviside mais elle à l'avantage d'être continue et dérivable.

Elle propose une transition douce de 0 à 1, la transition étant assez linéaire autour de 0.

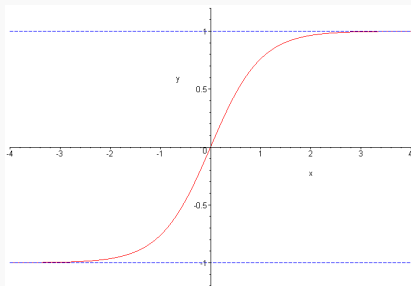
Elle permet de convertir la sortie z en une probabilité $\sigma(z)$

La fonction tangente hyperbolique.



La tangente hyperbolique est définie par :

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



Elle a des propriétés similaires à la fonction sigmoïde mais varie de -1 à +1.
C'est une fonction impaire : son graphe est symétrique par rapport à l'origine.



Ces fonctions d'activation semblent suivre une loi de Bernouilli

$$P(Y = y) = a(z)^y \times (1 - a(z))^{(1-y)}$$



Exercice 2

Pour chaque fonction activation, écrire une fonction qui renvoie sa valeur de sortie.



1. Donner une entrée au modèle.
2. Propager cette entrée à travers le réseau de neurones jusqu'à récupérer la sortie.

Une fois la sortie obtenue, nous pouvons la comparer à la sortie voulue et donc calculer une erreur. On ajuste les paramètres du modèle pour diminuer l'erreur précédemment calculée. Pour cela on soustrait à chaque paramètre la dérivée de l'erreur par rapport à lui-même (gradient descendant). On recommence à l'étape 1.

L'étape la plus importante est la 4^{ième}. Nous voulons être capable de créer autant de couches que l'on veut, de n'importe quel type, et d'utiliser n'importe quelle fonction d'activation. Seulement, en changeant l'architecture du réseau de neurones, on change également la formule littérale du calcul de la dérivée de l'erreur par rapport aux paramètres.



- ▶ Entraîner un neurone artificiel sur des données de références (X, y) pour que celui-ci renforce ses paramètres W a chaque fois qu'une entrée X est activée en même temps que la sortie y présente dans ces données.
- ▶ Les paramètres W sont mise a jour en calculant la différence entre la sortie de référence y et la sortie \hat{y} produite par le neurone en multipliant cette différence par X et par *alpha* vitesse apprentissage.

$$W = W - \alpha(y - \hat{y})X$$

- ▶ Y : sortie de reference
- ▶ \hat{y} sortie predite par le neurone
- ▶ X : entree du neurone
- ▶ α vitesse apprentissage (Learning rate)



1. Forward Propagation

$$z = w_1x_1 + w_2x_2 + b$$

$$a = \frac{1}{1 + e^{-z}}$$

2. Calcul de l'erreur (cost function)

$$L = -\frac{1}{m} \sum_{i=1}^m (y_i \log(a_i) + (1 - y_i) \log(1 - a_i))$$

3. Backward Propagation

4. Gradient descent

$$W = W - \alpha \frac{\sigma L}{\sigma W}$$



Une fonction Cout est une fonction qui permet de mesurer les erreurs effectuees par un modele.

Le Cost Function le plus utilise suivant la loi des entrées est :

Log Loss

$$L = -\frac{1}{m} \sum_{i=1}^m (y_i \log(a_i) + (1 - y_i) \log(1 - a_i))$$



La fonction de perte dans un réseau neuronal quantifie la différence entre le résultat attendu et le résultat produit par le modèle d'apprentissage automatique.

À partir de la fonction de perte, on peut dériver les gradients qui sont utilisés pour mettre à jour les poids. La moyenne de toutes les pertes constitue le coût.

Un modèle d'apprentissage automatique tel qu'un réseau neuronal tente d'apprendre la distribution de probabilité sous-jacente aux observations de données données.

Dans l'apprentissage automatique, nous utilisons généralement le cadre statistique de l'estimation du maximum de vraisemblance comme base de la construction du modèle.

Cela signifie essentiellement que nous essayons de trouver un ensemble de paramètres et une distribution de probabilité antérieure, telle que la distribution normale, pour construire le modèle qui représente la



Remarque

le choix de la fonction de perte est directement lié à la fonction d'activation utilisée dans la couche de sortie du réseau neuronal. Ces deux éléments de conception sont liés.

Considérez la configuration de la couche de sortie comme un choix sur le cadrage de votre problème de prédiction, et le choix de la fonction de perte comme la manière de calculer l'erreur pour un cadrage donné de votre problème.

Nous allons examiner les meilleures pratiques ou les valeurs par défaut pour chaque type de problème en ce qui concerne la couche de sortie et la fonction de perte.

Quelle fonction de perte utiliser ?



- ▶ **Problème de régression** $Y \in \mathbf{R}$: prédire une quantité de valeur réelle.
 - ▶ Couche de sortie : Activation linéaire
 - ▶ Fonction de perte : Erreur quadratique moyenne
- ▶ **Problème de classification binaire** $Y \in \{0, 1\}$: prédire la probabilité d'appartenir à une classe
 - ▶ couche de sortie : Activation sigmoïde.
 - ▶ Fonction de perte : Binary Cross-Entropy.
- ▶ **Problème de classification multi-classes** $Y \in \{0, 1, 2, \dots, m\}$: prédire la probabilité d'appartenir à chaque classe.
 - ▶ couche de sortie : Un nœud pour chaque classe utilisant la fonction d'activation softmax.
 - ▶ Fonction de perte : Categorical Cross-Entropy.





Consiste a déterminer comment la sortie du réseau varie en fonction des paramètres (W,b) présent dans chaque couche du modèle.
Pour ce faire on définit une chaîne de gradient qui indique comment la sortie varie en fonction de la dernière couche ainsi de suite

$$\frac{\delta Y}{\delta f_4}$$

$$, \frac{\delta f_4}{\delta f_3}$$



On peut mettre à jour les paramètres (W, b) de chaque couche de telle sorte à ce qu'ils minimisent l'erreur entre la sortie du modèle et la réponse attendue.

On utilise la descente de Gradient

$$W = W - \alpha \frac{\delta(y - \hat{y})}{\delta W}$$



La descente de Gradient (dérivé) consiste a ajuster les paramètres de façon a minimiser les erreurs du modèle ie minimiser la fonction Coût(Log Loss). il faut déterminer comment la fonction varie en fonction des différents paramètres. il faut que la fonction présente qu'un seul minimum (Fonction Convexe)



Pour trouver le minimum optimal, nous pouvons utiliser ces optimiseurs suivant: Nous avons vu que la fonction de perte présente plusieurs minima locaux qui peuvent induire notre modèle en erreur. Nous pouvons éviter que cela ne se produise si nous pouvons surveiller et fixer le taux d'apprentissage correctement.

Comme il n'est pas possible de le faire manuellement, Optimizers le fait pour nous. Il optimise automatiquement le taux d'apprentissage pour empêcher le modèle d'entrer dans un minimum local et est également responsable de l'accélération du processus d'optimisation.

- ▶ Adam
- ▶ Adagrad
- ▶ RMSProp
- ▶ SGD.



$$z = w_1 \times x_1 + w_2 \times x_2 + b = X.\text{dot}(W) + b$$

$$a = \frac{1}{1 + e^{-z}}$$

$$L = -\frac{1}{m} \sum_{i=1}^m y^i \log(a^i) + (1 - y^i) \log(1 - a^i)$$

$$w = w - \alpha \frac{\delta L}{\delta w}$$

$$b = b - \alpha \frac{\delta L}{\delta b}$$

$$\frac{\delta L}{\delta w_j} = \frac{1}{m} (a - y) x_j$$

$$\frac{\delta L}{\delta b} = \frac{1}{m} (a - y)$$



Le choix du Learning rate est fondamental pour la convergence de l'algorithme

- ▶ S'il est très élevé, les valeurs des poids seront modifiées dans une large mesure et la valeur optimale sera dépassée.
- ▶ S'il est très faible, les pas sont minuscules et il faut beaucoup de pas pour optimiser.



Chapitre 2 : Convolutional Neuronal Network



L'un des types de méthodes d'apprentissage profond les plus célèbres : les Réseaux Neuraux Convolutifs (parfois appelés ConvNets ou CNN). Ces réseaux traitent principalement des images.



Les images ne sont que des chiffres ! Contrairement à d'autres types de données, les images ont une structure spatiale ! le même objet peut être vu de différents points de vue, déformé, mis à l'échelle, occlus, etc...

Pour prendre en compte cette structure spatiale, nous avons besoin d'ajouter des informations spatiales à nos modèles. C'est là que se déroulent les réseaux neuronaux convolutifs (CNN) ! Alors, tout d'abord, parlons un peu du traitement de l'image.



Dans les algorithmes traditionnels de machine learning, la notion de features (caractéristiques) en vision sont utilisées pour faire de la classification d'images. Ces méthodes consistent à extraire les features de chaque image du jeu de données manuellement par un expert, puis à entraîner un classifieur sur ces features. Ces techniques d'apprentissage supervisé peuvent fournir de très bons résultats, et leur performance dépend fortement de la qualité des features préalablement trouvées. Il existe plusieurs méthodes d'extraction et de description de features.



Les réseaux de neurones convolutifs ont une méthodologie similaire à celle des méthodes traditionnelles d'apprentissage supervisé : ils reçoivent des images en entrée, détectent les features automatiquement de chacune d'entre elles, puis entraînent un classifieur dessus.

L'architecture spécifique du réseau permet d'extraire des features de différentes complexités, des plus simples au plus sophistiquées. L'extraction et la hiérarchisation automatiques des features, qui s'adaptent au problème donné, constituent une des forces des réseaux de neurones convolutifs.



L'architecture est alors plus spécifique : elle est composée de deux blocs principaux.

- 1 Le premier bloc fait la particularité de ce type de réseaux de neurones, puisqu'il fonctionne comme un extracteur de features. Pour cela, en appliquant des opérations de filtrage par convolution. La première couche filtre l'image avec plusieurs noyaux de convolution, et renvoie des "feature maps", qui sont ensuite normalisées (avec une fonction d'activation) et/ou redimensionnées. Ce procédé peut être réitéré plusieurs fois : on filtre les features maps obtenues avec de nouveaux noyaux, ce qui nous donne de nouvelles features maps à normaliser et redimensionner, et qu'on peut filtrer à nouveau, et ainsi de suite. Finalement, les valeurs des dernières feature maps sont concaténées dans un vecteur. Ce vecteur définit la sortie du premier bloc, et l'entrée du second.



- 2 Le second bloc : Les valeurs du vecteur en entrée sont transformées (avec plusieurs combinaisons linéaires et fonctions d'activation) pour renvoyer un nouveau vecteur en sortie. Ce dernier vecteur contient autant d'éléments qu'il y a de classes : l'élément i représente la probabilité que l'image appartienne à la classe i



Il existe quatre types de couches pour un réseau de neurones convolutif :

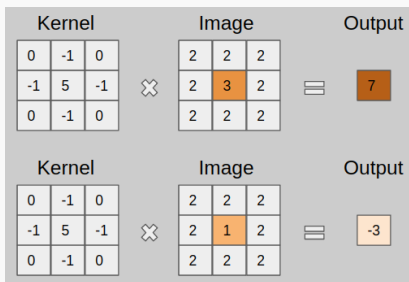
1. la couche de convolution,
2. la couche de pooling,
3. a couche de correction ReLU
4. la couche fully-connected.



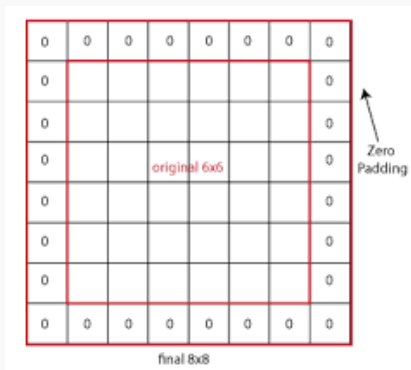
- ▶ Une convolution est une opération mathématique sur une image avec un noyau donné, résultant en une nouvelle image traitée.
- ▶ La convolution, d'un point de vue simpliste, est le fait d'appliquer un filtre mathématique à une image.
- ▶ D'un point de vue plus technique, il s'agit de faire glisser une matrice par-dessus une image, et pour chaque pixel, utiliser la somme de la multiplication de ce pixel par la valeur de la matrice.

Cette technique nous permet de trouver des parties de l'image qui pourraient nous être intéressantes.

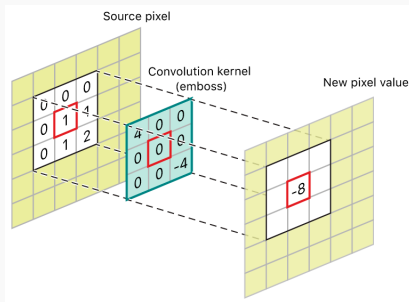
Convolution



Le Padding est une methode qui ajoute des bordures a une image, en augmentant ainsi la taille de l'image. La plupart du temps, les images sont rembourrées de zéros.



Un noyau dans le traitement de l'image n'est rien de plus qu'une petite image (ou une petite matrice), que nous utiliserons pour faire une convolution. Habituellement, les noyaux ont des dimensions impaires et sont assez petits.

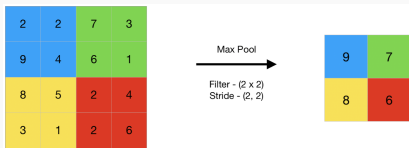




Ce type de couche est souvent placé entre deux couches de convolution : elle reçoit en entrée plusieurs feature maps, et applique à chacune d'entre elles l'opération de Pooling.

L'opération de Pooling consiste à réduire la taille des images, tout en préservant leurs caractéristiques importantes.

Pour cela, on découpe l'image en cellules régulières, puis on garde au sein de chaque cellule la valeur maximale (la valeur minimale ou la moyenne ou tout autre valeur suivant une technique donnée).





La couche de Pooling permet de réduire le nombre de paramètres à calculer dans le réseau. On améliore ainsi l'efficacité du réseau et on évite le sur-apprentissage.

La couche de Pooling rend le réseau moins sensible à la position des features : le fait qu'une feature se situe un peu plus en haut ou en bas, ou même qu'elle ait une orientation légèrement différente ne devrait pas provoquer un changement radical dans la classification de l'image.

La méthode utilisée consiste à imaginer une fenêtre de 2 ou 3 pixels qui glisse au-dessus d'une image, comme pour la convolution. Mais, cette fois-ci, nous faisons des pas de 2 pour une fenêtre de taille 2, et des pas de 3 pour 3 pixels. La taille de la fenêtre est appelée « kernel size » et les pas s'appellent « strides » .

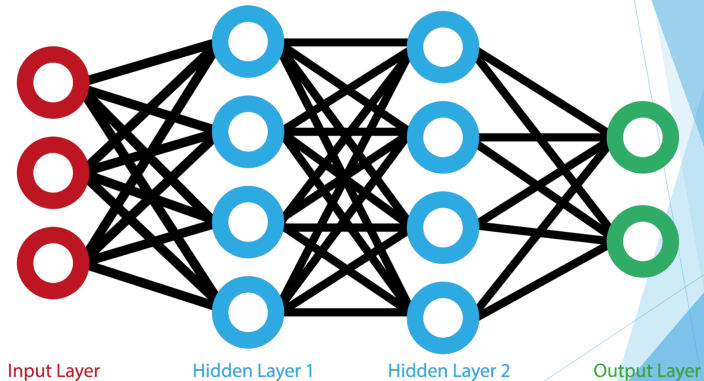


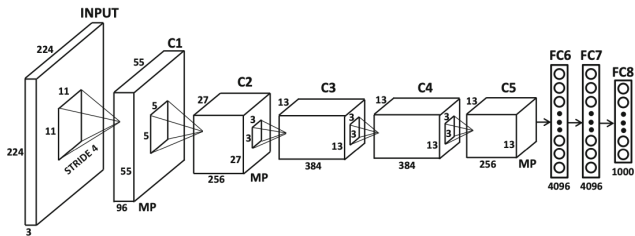
ReLu est une fonction qui doit être appliquée à chaque pixel d'une image après convolution, et remplace chaque valeur négative par un 0. ReLu est très utilisée dans les réseaux de neurones à convolution car il s'agit d'une fonction rapide à calculer :

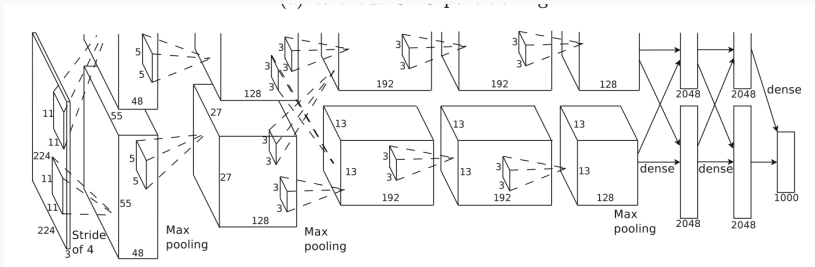
$$g(x) = \begin{cases} 0 & \text{si } z < 0 \\ z & \text{si } z \geq 0 \end{cases}$$

Sa performance est donc meilleure que d'autres fonctions où des opérations coûteuses doivent être effectuées.

Fully Connected









Chapitre 3 : Recurrent Neuronal Network



Ils sont utilisés dans de nombreuses applications modernes nécessitant de traiter des **séquences d'informations**.

Vous jouez tous les jours avec des informations séquentielles, sans même vous en rendre compte !

Il existe donc un certain nombre de problèmes séquentiels :

- ▶ traduction linguistique
- ▶ reconnaissance vocale.
- ▶ classification de textes de sentiment
- ▶ génération de textes / de musiques
- ▶ traducteur de langue automatique



Prenons un cas simple. Nous avons le problème suivant :

"This morning I took my cat for a walk."

given these words predict the next word

Comment le résoudriez-vous en utilisant l'apprentissage automatique ?



Idée 1 : Faire un encode et appliquer un MLP

C'est une bonne idée, mais que se passe-t-il si nous avons des dépendances à long terme, comme dans une phrase telle que la suivante :

J'ai vécu en Chine quand j'étais enfant, même si je suis français. C'est pourquoi je parle couramment...

Est-ce que cela pourrait s'appliquer à ce type de phrase ? Ne prédit-elle pas que le gars parle français alors qu'elle est censée prédire le chinois ?

"This morning I took my cat for a walk."

given these predict the
two words next word

One-hot feature encoding: tells us what each word is

[1 0 0 0 0 0 1 0 0 0]

for

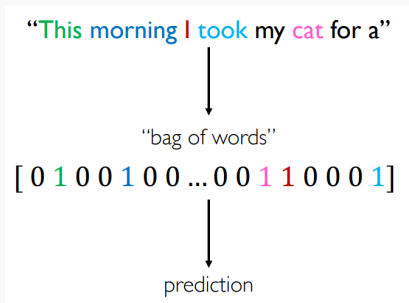
a



prediction



Idée 2 : Appliquer un encodage BOW (ou TF-IDF) :
BOW et TF-IDF sont des fonctions très puissantes.
Mais elles ne préservent pas l'ordre de la séquence.
Exemple pour un BOW, les deux phrases suivantes sont identiques :
La nourriture est bonne, pas mauvaise du tout !
La nourriture est mauvaise, pas bonne du tout !





Les réseaux de neurones « standard » ?

- ▶ Ne tient pas compte des différentes positions des mots

Mais les reseaux de neurones recurrents nous avons :

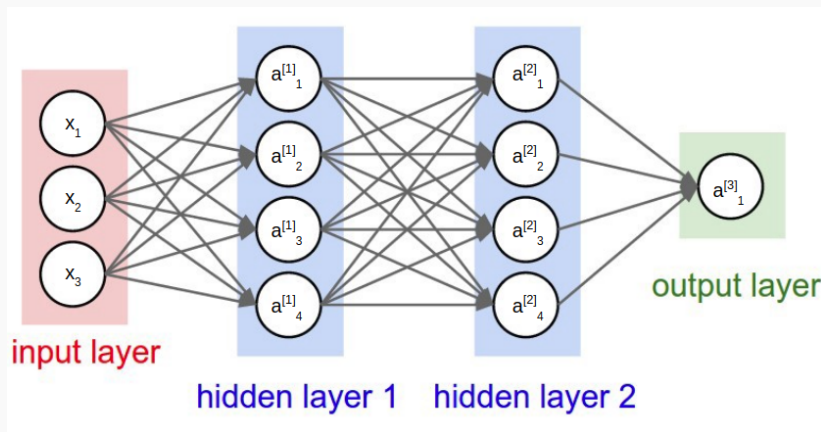
- ▶ possibilité de manipuler des séquences de taille variable
- ▶ tout calcul futur tient compte des calculs passés
- ▶ les poids / paramètres sont partagés dans le temps



Pour traiter correctement les informations séquentielles, voici ce qu'il faudrait :

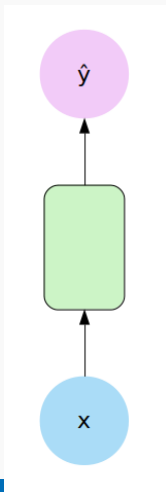
- ▶ gérer des séquences de longueur variable
- ▶ suivre les dépendances à long terme
- ▶ conserver des informations sur l'ordre
- ▶ partager les paramètres sur l'ensemble de la séquence

Jusqu'à présent, vous avez l'habitude de voir des diagrammes dans lesquels les réseaux neuronaux vont de gauche (avec les caractéristiques) à droite (avec la prédiction).





Lorsque nous utilisons des réseaux neuronaux récurrents, nous remplaçons cette représentation par la suivante : En bleu, les caractéristiques d'entrée ; en vert, les couches du réseau neuronal ; en violet, la prédiction.





En fonction du problème à résoudre, de nombreux types de RNN peuvent être utilisés.

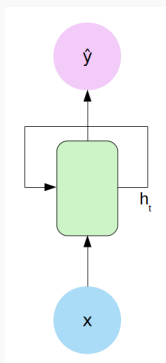
1. une analyse des sentiments (un seul résultat)
2. une traduction linguistique (plusieurs résultats)

ont des exigences différentes.

1. one-to-one est un MLP tel que vous le connaissez déjà
2. many-to-many pourrait être un modèle de traduction : il entre une séquence de mots en anglais et sort une séquence de mots en français.
3. many-to-one pourrait être un modèle d'analyse des sentiments : il entre une séquence de mots et sort une critique.

un RNN se calcule en deux étapes.

1. vous calculez un **état caché** h_t en utilisant à la fois
 - ▶ les caractéristiques d'entrée x_t
 - ▶ l'état caché précédent h_{t-1}
2. vous calculez la prédiction \hat{y} en utilisant cet état caché h_t .



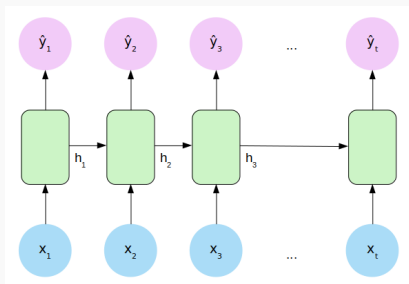


Comment un RNN calcule-t-il les prédictions ?

Nous considérons que

- ▶ $x_1, x_2, x_3 \dots x_t$ sont les mots numéro 1, 2, 3... t d'une phrase en anglais.
- ▶ Les cibles $y_1, y_2, y_3 \dots y_t$ sont les mots numéro 1, 2, 3... t de la même phrase en français.

Notre réseau neuronal ressemblera donc à ceci :

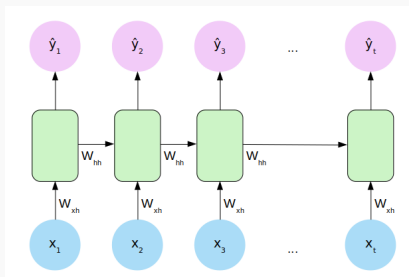


Dans un RNN, les mêmes poids sont partagés pour toutes les étapes de la séquence :

Les poids W_{xh} et W_{hh} permettent de calculer l'état caché comme le ferait un Perceptron :

$$h_t = g(W_{hh}h_{t-1} + W_{xh}X_t + b)$$

C'est exactement comme un Perceptron, ou un réseau neuronal classique, où $W_{xh}X_t$ est une somme pondérée des caractéristiques X_t , et g est simplement une fonction d'activation.

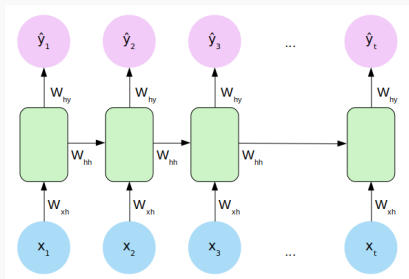




La deuxième étape consistera à calculer les prédictions \hat{y} . C'est ici qu'apparaissent les poids W_{hy} .

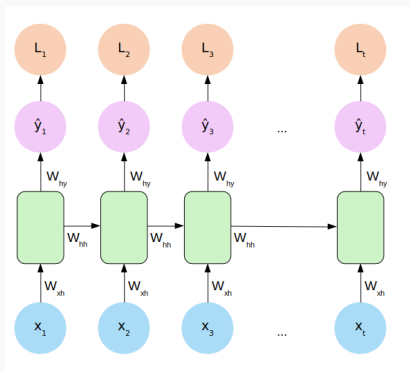
En utilisant ces poids, cela fonctionnera comme un Perceptron :

$$\hat{y}_t = g(W_{hy}h_t + b)$$

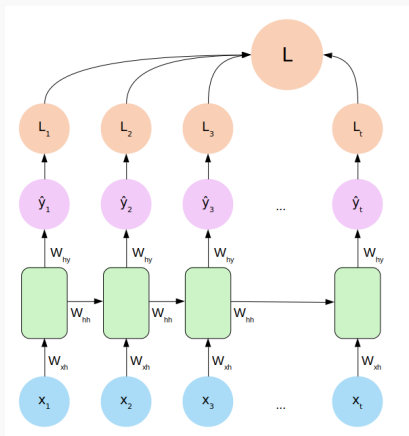


- ▶ Comment calculer la perte dans un réseau aussi compliqué ?
- ▶ Comment la minimise avec la descente de gradient ?

Chaque pas de temps est un réseau neuronal ordinaire (un MLP). On calcule une perte pour chaque pas de temps,



A chaque étape t , nous obtenons une perte L_t . Il suffit ensuite de les additionner pour obtenir la perte globale !





Les données IMDB est un jeu de données contenant des critiques de films, ainsi qu'une étiquette associée 0 (critique négative) ou 1 (critique positive).

	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Matter's "Love in the Time of Money" is...	positive
5	Probably my all-time favorite movie, a story o...	positive
6	I sure would like to see a resurrection of a u...	positive
7	This show was an amazing, fresh & innovative i...	negative
8	Encouraged by the positive comments about this...	negative
9	If you like original gut wrenching laughter yo...	positive

```
1 import numpy as np
2 from tensorflow.keras import datasets
3 imdb = datasets.imdb
4
5 (X_train, y_train), (X_test, y_test) = imdb.load_data(num_words
    =10000)
6
7 X_train.shape, y_train.shape
8
9
10 from tensorflow.keras.preprocessing import sequence
11
12 X_train = sequence.pad_sequences(X_train,
13                                 value=0,
14                                 padding='post', # to add zeros
15                                 maxlen=256) # the length we
16                                 want
17
18 X_test = sequence.pad_sequences(X_test,
19                                value=0,
20                                padding='post', # to add zeros
21                                maxlen=256) # the length we want
```

Listing 1: Preprocessing

```
1 from tensorflow.keras.models import Sequential
2 from tensorflow.keras.layers import SimpleRNN, Dense, Embedding
3
4
5
6 def my_RNN():
7
8     model = Sequential()
9     # The input_dim is the number of different words we have in
10    # our corpus: here 10000
11    # The input_length is the length of our sequences: here 256
12    # thanks to padding
13    model.add(Embedding(input_dim=10000, output_dim=32,
14                        input_length=256))
15
16    # We add two layers of RNN
17    model.add(SimpleRNN(units=8, return_sequences=True))
18    model.add(SimpleRNN(units=8, return_sequences=False))
19
20    # Finally we add a sigmoid
21    model.add(Dense(units=1, activation='sigmoid'))
22
23    return model
```

Listing 2: Building a RNN



Plusieurs choses peuvent être remarquées ici :

- ▶ Couche **Embedding** : elle convertit notre entrée BOW en Word Embedding comme des nombres, donc ici fondamentalement chaque mot est transformé en un tableau de 32 caractéristiques.
- ▶ Nous avons empilé **deux couches RNN** : cela ne signifie pas que nous faisons deux RNN, cela signifie simplement que notre réseau neuronal contient deux couches.
 - ▶ le premier RNN a **return_sequences=True** : la séquence est nécessaire quand une autre couche de 'RNN' est ajoutée.
 - ▶ le second a 'return_sequences=False' :

La règle empirique pour many-to-one est : return_sequences=True quand il y a une autre couche RNN et return_sequences=False sinon.

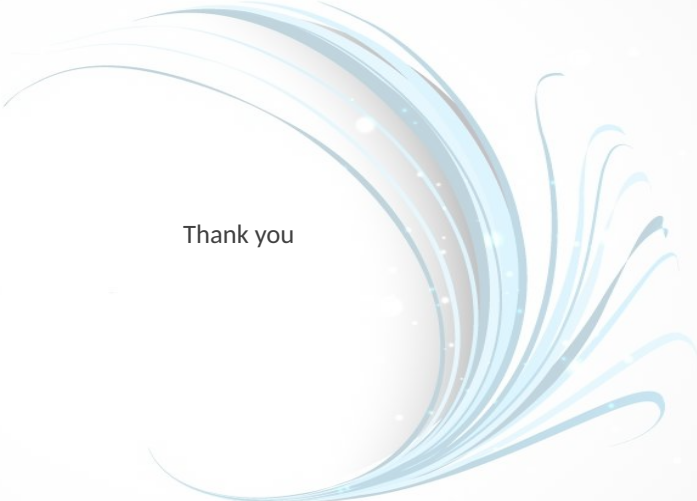
```
1 model = my_RNN()
2
3
4
5 model.compile(optimizer='rmsprop',
6               loss='binary_crossentropy',
7               metrics=['accuracy'])
8
9
10
11 model.fit(x=X_train, y=y_train, validation_data=(X_test, y_test)
12          , epochs=10, batch_size=128)
13
14 from sklearn.metrics import accuracy_score
15
16 print('accuracy on train with NN:', model.evaluate(X_train,
17            y_train)[1])
18 print('accuracy on test with NN:', model.evaluate(X_test, y_test
19            ) [1])
```

Listing 3: Compilation



Suite :

1. **Advanced Recurrent Neuronal Network**
 - ▶ GRU
 - ▶ LSTM
2. **Generative Adversarial Networks and Adversarial Training (GAN)**
3. **AutoEncoder**



Thank you