Réseaux de neurones convolutionnels

Introduction à l'apprentissage automatique
Master Sciences Cognitives
Aix Marseille Université

Alexis Nasr

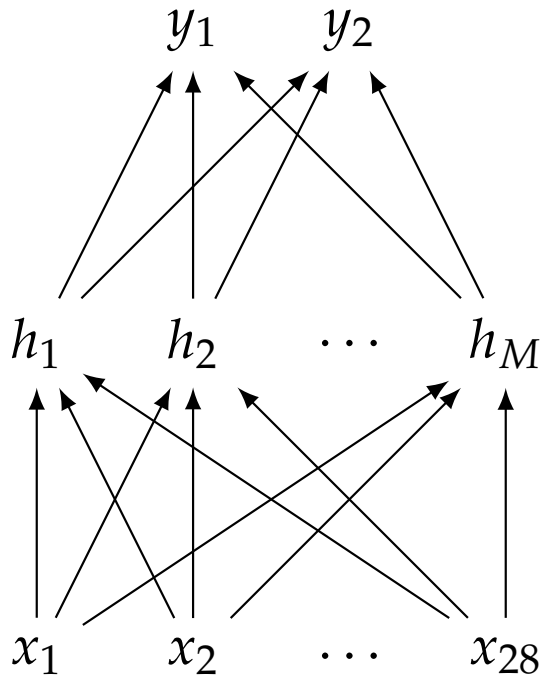
Motivations

- Certains problèmes de prédiction portent sur des objets possédant une structure unidimensionnelle (mots dans une phrases, mesures à intervalles réguliers ...) ou bidimensionnelle (image).
- Ces objets possèdent souvent des **motifs locaux** pertinents (séquences de deux ou trois mots, petites zones dans une image).
- Dans certains cas, la position exacte de tels motifs dans l'objet à traiter n'est pas très importante, en revanche leur présence l'est.
- Le perceptron multicouche n'est pas adapté pour traiter ce type d'objet.
- Les réseaux convolutionnels sont plus adaptés pour prendre en compte ces caractéristiques.

Objectifs

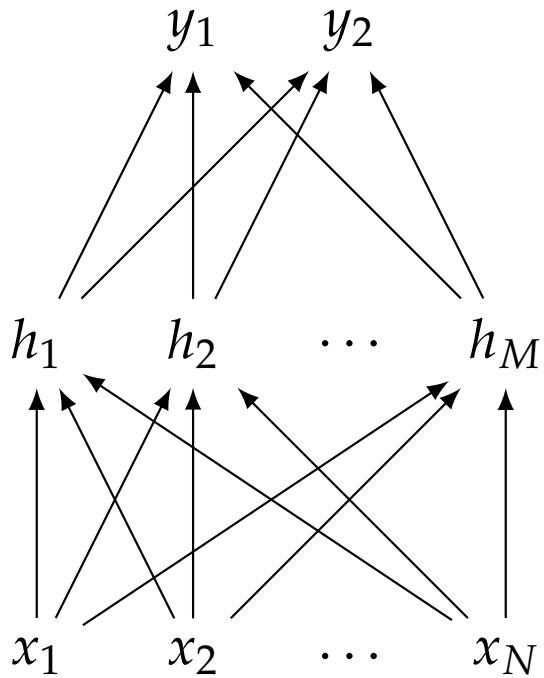
- Comprendre la notion de partage de paramètres
- Comprendre la notion de faible connectivité
- Comprendre les notions de couches de convolution et de couches de pooling
- Savoir construire un réseau convolutionnel avec `keras`

Retour sur la prédiction de la langue



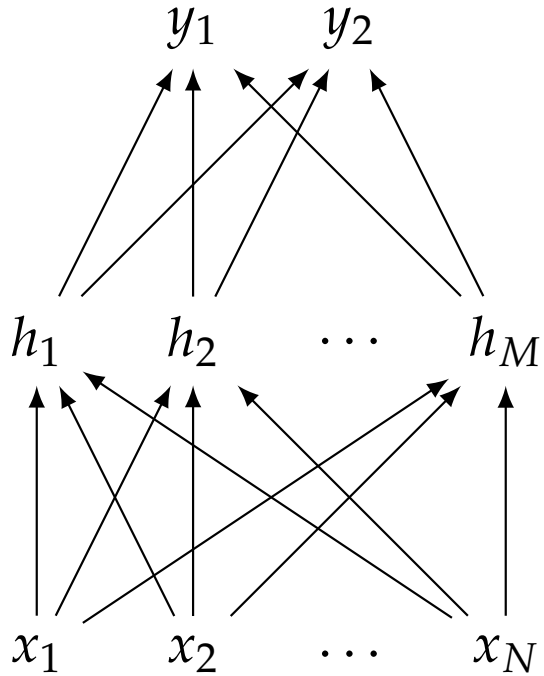
- Perceptron multicouche
- $x_1 \ldots x_{28}$ sont les fréquences des lettres dans un document.
- Chaque neurone de la couche cachée est lié à toutes les entrées : matrice $28 \times M$.
- Serait il possible de partir directement du texte ?

Retour sur la prédiction de la langue



- $x_1 \ldots x_N$ correspondent aux lettres dans un document de longueur N
- La matrice de la couche cachée est de dimension $N \times M$

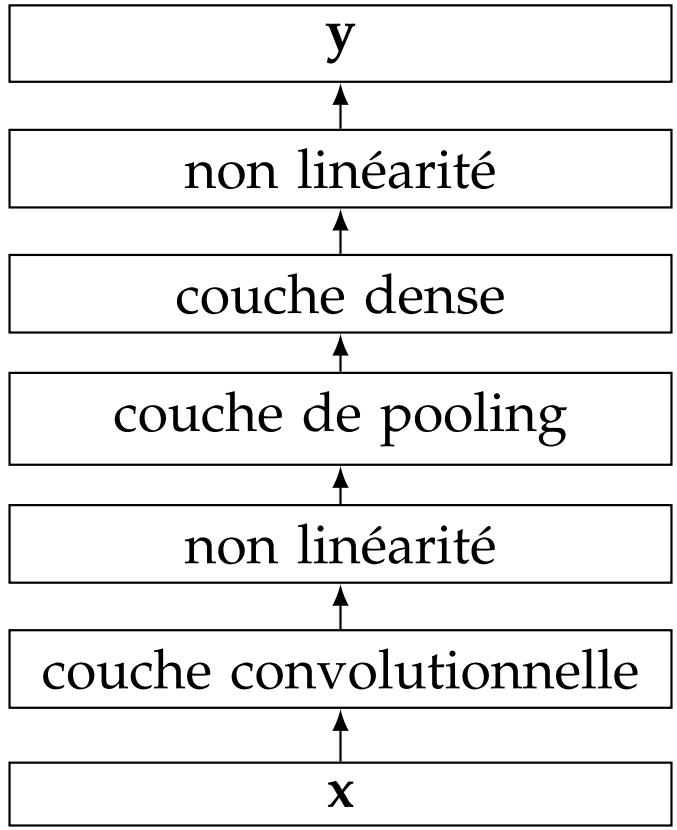
Retour sur la prédiction de la langue



Le modèle n'est pas très satisfaisant :

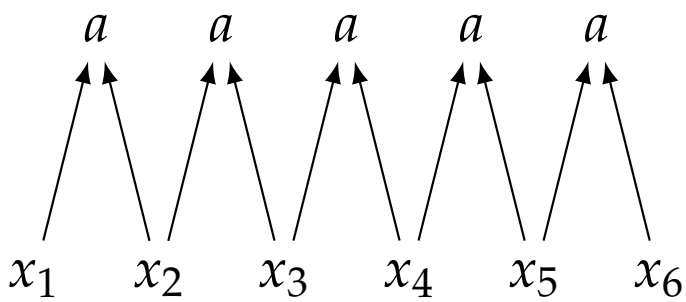
- Le modèle traite différemment une même lettre selon la position qu'elle occupe dans le texte.
- Les motifs intéressants sont probablement locaux (bigrammes, trigrammes …).
- Or le modèle *voit* toute la séquence, ce qui n'est probablement pas très utile et nécessite beaucoup de paramètres.
- Il faut un modèle différent pour chaque longueur de texte (N).

Réseau convolutionnel



Les réseaux convolutionnels peuvent prendre des formes très variées, on retrouve souvent une succession de couches de **convolution** et de **pooling**, suivie d'une couche **dense** pour la décision finale.

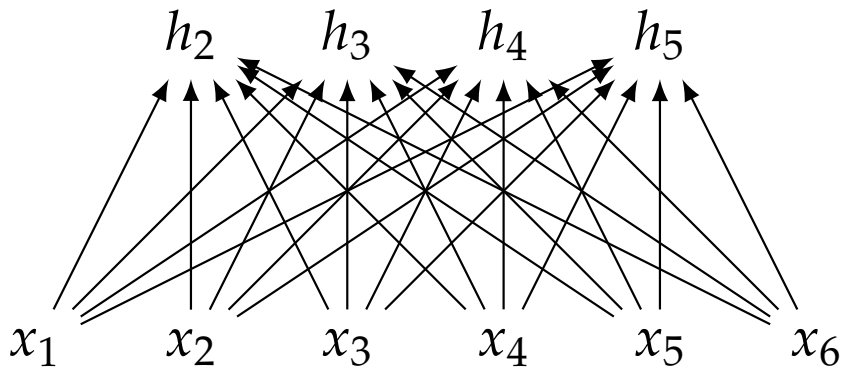
Couche de convolution



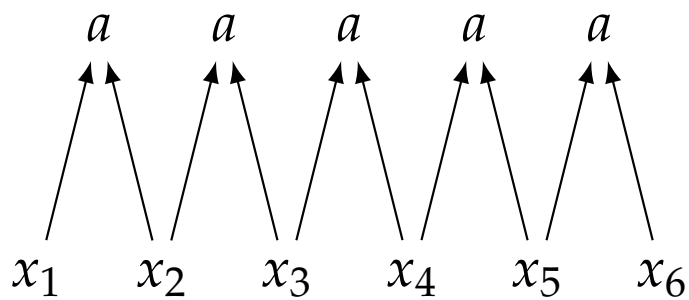
- Constituée de **copies** d'un même neurone (a).
- Chaque neurone ne *voit* qu'une partie de l'entrée (ici deux lettres).
- Deux propriétés importantes :
  - **faible connectivité** : Chaque neurone de la couche cachée n'est plus connecté à tous les éléments de la couche d'entrée.
  - **partage de paramètres** : Les mêmes paramètres sont utilisés pour chaque position du texte.
    Ce qui est appris pour une position donnée est aussi valable pour les autres positions.

Faible connectivité

- Perceptron multicouche : 4×6 paramètres.

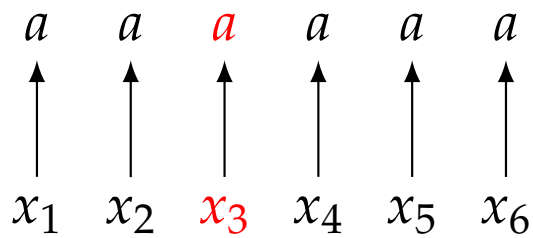


- Couche convolutionnelle : 2 paramètres

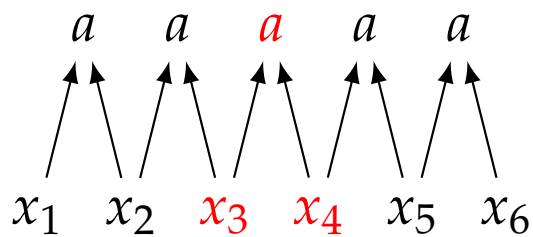


Noyau (Kernel)

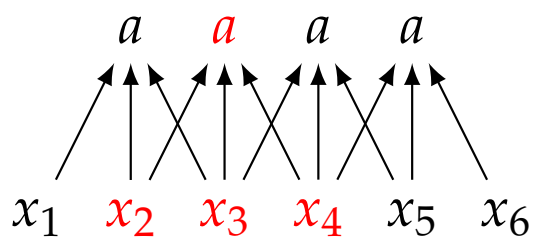
- Le nombre de connections d'un neurone convolutionnel est appelé **noyau**.
- Il correspond à la partie de la séquence vue par un neurone.
- Noyau = 1



- Noyau = 2

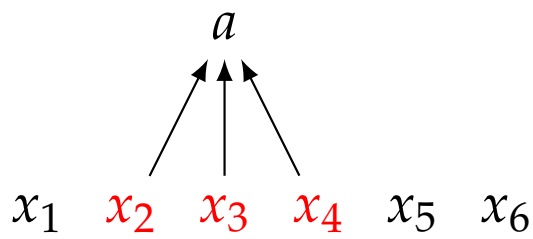


- Noyau = 3

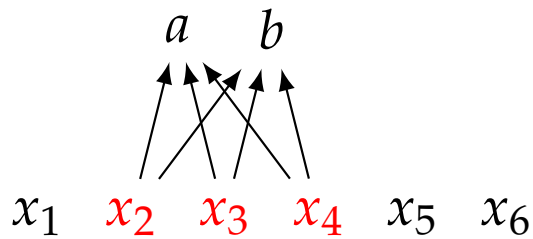


Canaux (Filters)

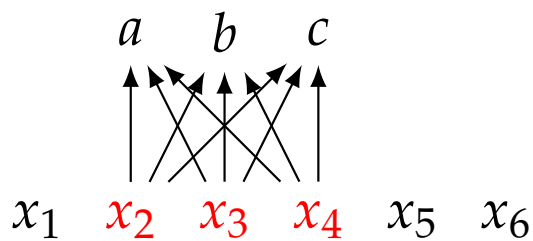
- Il est souvent nécessaire de multiplier le nombre de neurones liés aux mêmes entrées.
- Cela définit le nombre de **canaux** de la couche convolutionnelle.
- Un canal



- Deux canaux

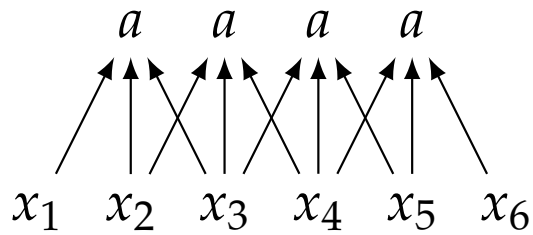


- Trois canaux

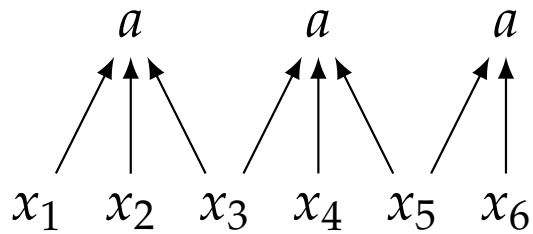


Pas (stride)

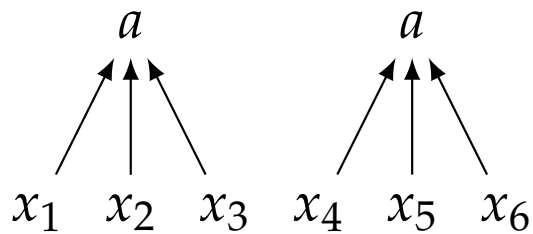
- Le segment de l'entrée vu par un seul neurone peut être vu comme une **fenêtre glissante** sur la séquence d'entrée.
- Le **pas** correspond au déplacement de la fenêtre glissante à chaque étape.
- Pas = 1



- Pas = 2

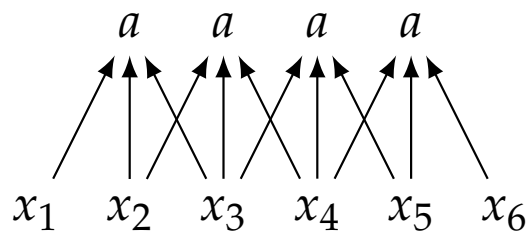


- Pas = 3



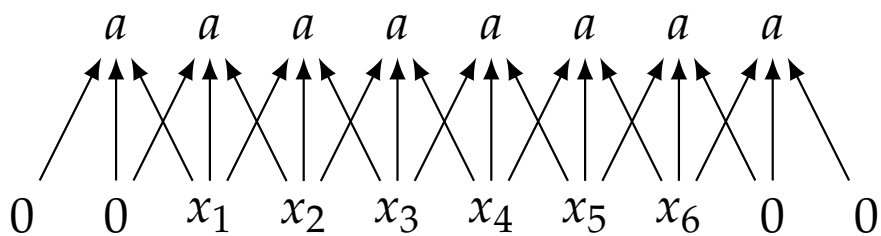
Rembourrage (Padding)

- Que se passe-t-il aux extrémités de la couche d'entrée ?
- Deux possibilités :
  - pas de padding : les premiers et les derniers éléments sont vus moins souvent que les autres.



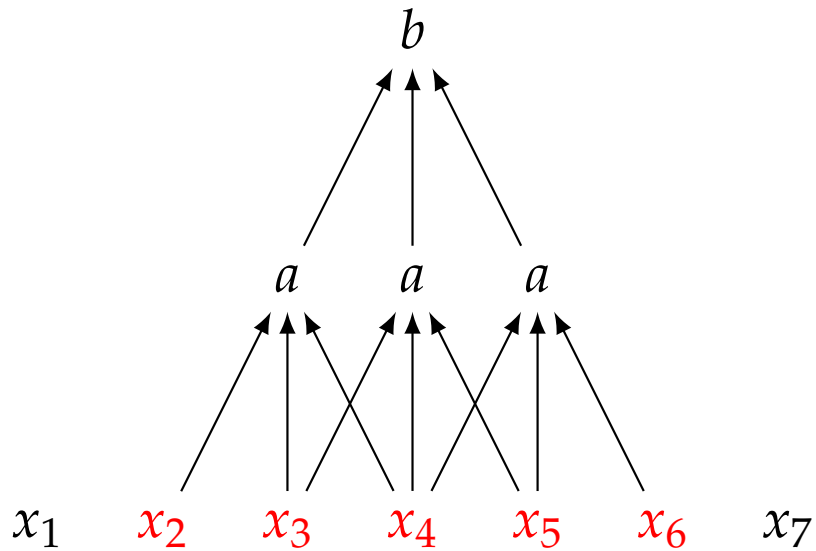
$$(x_1, x_2, x_3), (x_2, x_3, x_4), (x_3, x_4, x_5)$$

  - padding : on ajoute des éléments en plus à gauche et à droite.



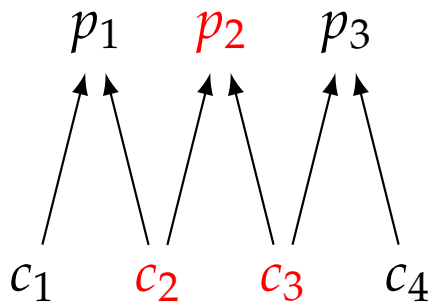
$$(0, 0, x_1), (0, x_1, x_2), (x_1, x_2, x_3)$$
$$(x_4, x_5, x_6), (x_5, x_6, 0), (x_6, 0, 0)$$

Empilement de couches de convolution



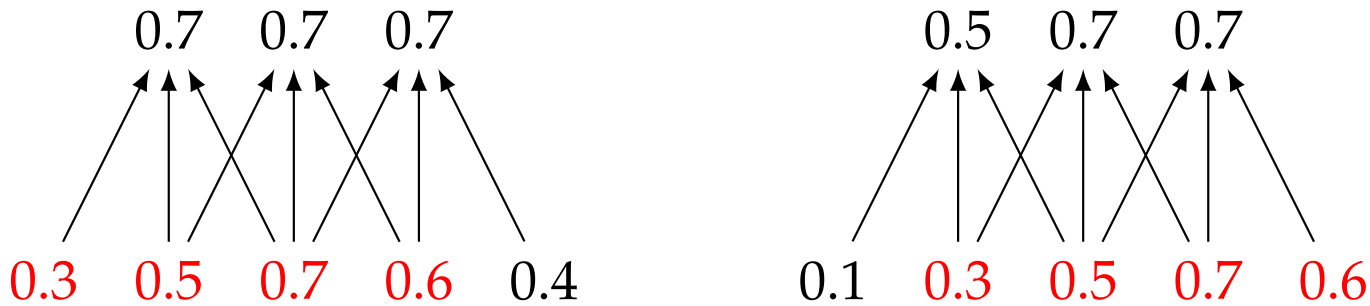
- Il est possible d'empiler les couches de convolution : la sortie d'une couche est l'entrée de la couche suivante.
- Au fur et à mesure qu'on empile les couches, on *voit* une partie plus importante de l'entrée.
- La partie de l'entrée vue est appelée **champ perceptif**.

Pooling



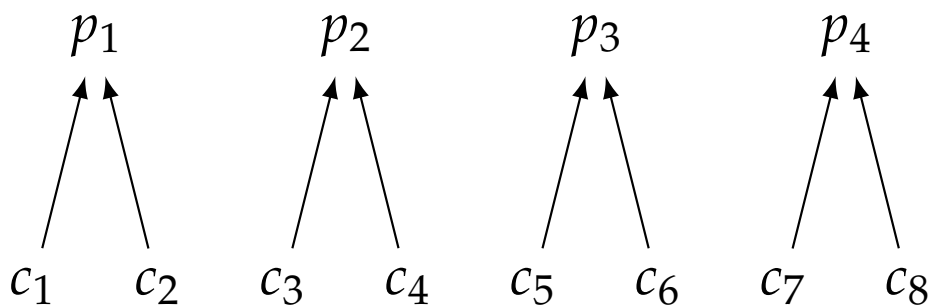
- Dans certains cas, on souhaite que la réponse du réseau change très peu lorsque l'entrée subit une légère translation.
- Pour cela on introduit une couche de **pooling** qui résume l'information portée par des neurone adjacents de la couche de convolution.
- Plusieurs choix sont possibles :
  - Max pooling : $p_2 = \max(c_2, c_3)$
  - Average pooling : $p_2 = 0.5 \times c_2 + 0.5 \times c_3$
- Les couches de pooling ne possèdent pas de paramètres dont il faut estimer la valeur.

Max pooling : exemple



- L'entrée a subi une translation d'un pas vers la droite.
- La couche de pooling a peu changé.
- La faible sensibilité à une légère translation est utile lorsqu'on s'intéresse plus à la présence d'une certaine caractéristique en entrée, qu'à sa position exacte.
- Dans un réseau on alterne généralement les couches de convolution et de pooling.

Sous-échantillonage (Downsampling)



- Le pooling peut être utilisé pour sous-échantillonner (*downsampling*) les sorties de la couche de convolution.
- La taille de la couche après pooling est divisée par la taille du noyau de la couche de pooling.

Implémentation avec keras

- Couche de convolution

```
tf.keras.layers.Conv1D(
    filters,          # nombre de canaux
    kernel_size,      # taille du noyau
    strides=1,        # taille du pas
    padding="valid", # avec ou sans padding
    use_bias=True,    # ajout d'un biais
    ...
)
```

- Couche de pooling

```
tf.keras.layers.MaxPooling1D(
pool_size=2,     # taille de la fenetre de pooling
strides=None,    # par defaut, valeur de pool_size (sous-échantillonage)
padding="valid",
...
)
```

Réseau convolutionnel pour l'identification de la langue

```
nbLangues = len(codeLangue.keys())
nbChar = 256
filtersNb = 32
kernelSize = 4

model = tf.keras.Sequential()
model.add(tf.keras.layers.Conv1D(filtersNb, kernelSize,input_shape=(nbChar, 28)))
model.add(tf.keras.layers.Activation(tf.keras.activations.relu))
model.add(tf.keras.layers.MaxPooling1D(pool_size=4))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=nbLangues))
model.add(tf.keras.layers.Activation(tf.keras.activations.softmax))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=50, batch_size = 32, validation_split=0.2)
```

Structure du réseau

```
Layer (type)                 Output Shape              Param #
=================================================================
conv1d (Conv1D)              (None, 256, 32)           3616
_________________________________________________________________
activation (Activation)      (None, 256, 32)           0
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 64, 32)            0
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 37)                75813
_________________________________________________________________
activation_1 (Activation)    (None, 37)                0
=================================================================
Total params: 79,429
Trainable params: 79,429
Non-trainable params: 0
```

- $3616 = 28 \times 4 \times 32 + 32 =$
  $28 \times \texttt{kernelSize} \times \texttt{filtersNb} + \texttt{filtersNb}(\textit{biais})$
- $75813 = 2048 \times 37 + 37$

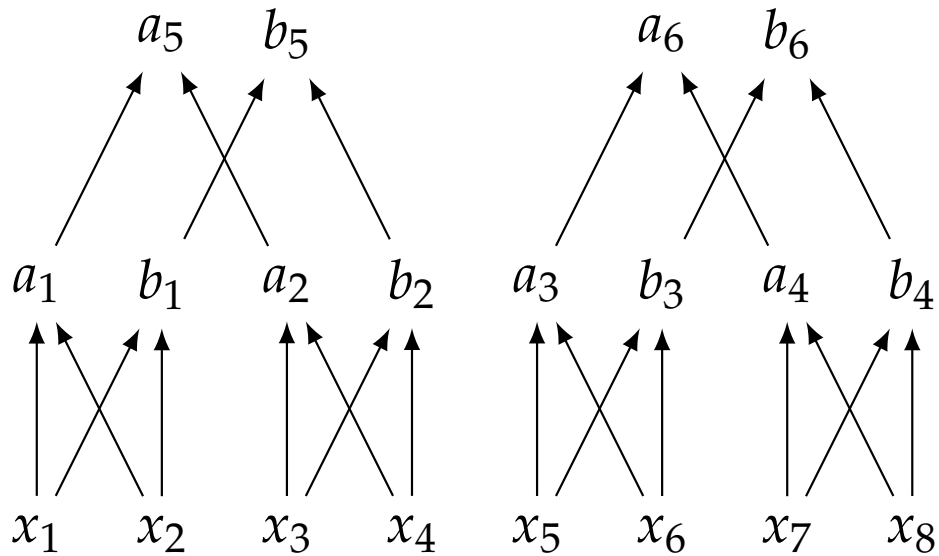
Format de l'entrée

- Les caractères composant le texte sont représentés par des entiers $c : 0 \leq c \leq 27$
- L'association (caractère $\leftrightarrow$ code) est **arbitraire**.
- Lorsque x_i représentait la fréquence de la i-ème lettre de l'alphabet, cela avait un sens de faire : $w_i \times x_i$
- Plus w_i est élevé, plus la fréquence de la i-ème lettre de l'alphabet est une feature importante.
- Mais si x_i est le code du i-ème caractère, cela n'a pas de sens de faire : $w_i \times x_i$
- Dans le premier cas x_i est une variable numérique, dans le second, c'est une variable catégorielle.

Format de l'entrée

- On adopte une représentation *one-hot* : un caractère est représenté par un vecteur de taille 28 dont la i-ème composante vaut 1 et les autres 0.
- La couche d'entrée du réseau se présente comme une liste de `nbChar` caractères, chaque caractère étant lui même une liste de 28 entiers (des 0 ou des 1), d'où : `input_shape=(nbChar, 28)`
- Pratiquement, la couche d'entrée se présente comme une liste de listes :
  `[[0,0,1,0],[1,0,0,0],[0,1,0,0]]`

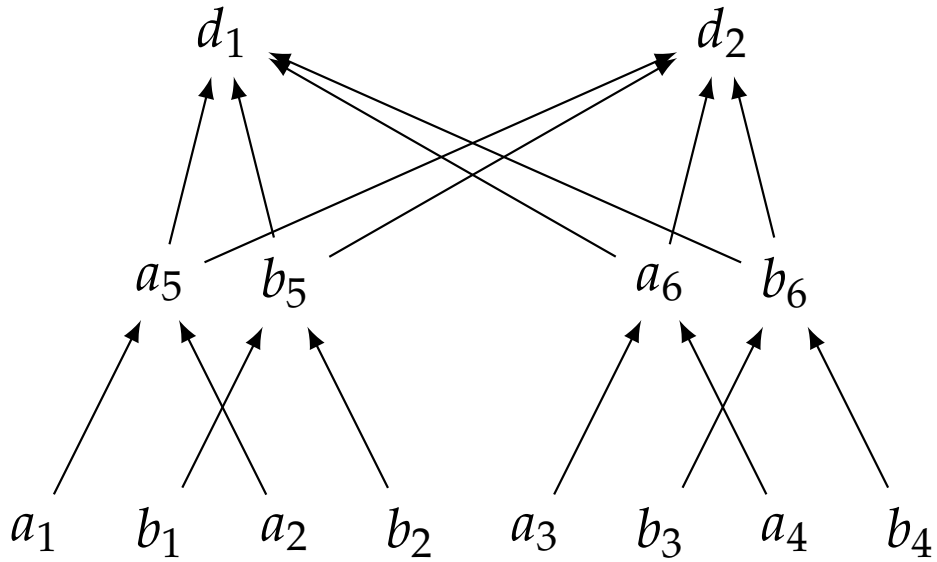
Structure des couches internes



taille du noyau = 2, pas = 2, canaux = 2

- Les couches de convolution sont composées de plusieurs canaux indépendants.
- Chaque canal peut être vu comme un réseau autonome (on ne mélange pas les a et les b).
- là aussi, on a des listes de listes :
  `[[a1, a2, a3, a4], [b1, b2, b3, b4]]`
  `[[a5, a6], [b5, b6]]`

Aplatissment



- Au niveau de la couche dense (d_1, d_2), les informations des différents canaux sont fusionnés pour prendre la décision finale.
- Opération d'aplatissement (flattening) :
  `[[a5, a6], [b5, b6]]` $\rightarrow$ `[a5, a6, b5, b6]`

Images

- Les réseaux convolutionnels vus jusque là prenaient en entrée des **séquences**.
- Ils peuvent aussi prendre en entrée des tableaux à deux dimension, comme dans le cas d'images.
- Au lieu de *voir* des segments de séquences, les couches de convolution *voient* des rectangles de pixels.

Implémentation avec keras

- Couche de convolution

```
tf.keras.layers.Conv2D(
filters,          # nombre de canaux
kernel_size,      # entier ou liste de deux entiers : hauteur et largeur du noyau
strides=(1, 1),   # deplacement en hauteur et en largeur
padding="valid", # avec ou sans padding
use_bias=True,    # avec ou sans biais
...
)
```

- Couche de pooling

```
tf.keras.layers.MaxPooling2D(
pool_size=(2, 2), # taille de la fenêtre de pooling
strides=None,     # par défaut pool_size (sous-échantillonage)
padding="valid",
...
)
```

Reconnaissance de chiffres manuscrits

- Tâche : reconnaître des chiffres manuscrits
  - Entrée : image de chiffres manuscrits
  - Sortie : 10 classes : 0, 1, . . . , 9
- Un chiffre est représenté par une image de 28×28 pixels dont la valeur indique le niveau de gris (0 (blanc) à 255 (noir)).

Exemple

```
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   3   18  18  18  126 136 175 26  166 255 247 127 0   0   0   0
0   0   0   0   0   0   0   0   30  36  94  154 170 253 253 253 253 253 225 172 253 242 195 64  0   0   0   0
0   0   0   0   0   0   0   49  238 253 253 253 253 253 253 253 253 253 251 93  82  82  56  39  0   0   0   0
0   0   0   0   0   0   0   18  219 253 253 253 253 253 253 198 182 247 241 0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   80  156 107 253 253 205 11  0   43  154 0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   14  1   154 253 90  0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   139 253 190 2   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   11  190 253 70  0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   35  241 225 160 108 1   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   81  240 253 253 119 25  0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   45  186 253 253 150 27  0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   16  93  252 253 187 0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   249 253 249 64  0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   46  130 183 253 253 207 2   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   39  148 229 253 253 253 250 182 0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   24  114 221 253 253 253 253 201 78  0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   23  66  213 253 253 253 253 19  8   81  2   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   18  171 219 253 253 253 253 195 80  9   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   55  172 226 253 253 253 253 244 133 11  0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   136 253 253 253 212 135 132 16  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Modèle avec keras

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Flatten, Activation
from keras.layers import Conv2D, MaxPooling2D

model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(2,2), strides=(1,1), input_shape=(28,28,1)))
model.add(Activation(keras.activations.relu))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
model.add(Flatten())
model.add(Dense(10))
model.add(Activation(keras.activations.softmax))
```

- Encodage de l'entrée :
  - liste de 28 lignes chacune composée de 28 valeurs (0 à 255)
  - `input_shape=(28,28,1)`
- Encodage de la sortie :
  - représentation *one-hot* : $2 \leftrightarrow$ `[0,0,1,0,0,0,0,0,0,0]`

Sources

- Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep Learning*, MIT Press, 2016.