Indexation et Recherche Multimedia par Deep Learning

Polytech Grenoble - INFO4

Philippe Mulhem

Groupe Modélisation et Recherche d'Information Multimédia

Laboratoire d'Informatique de Grenoble

Plan

- Introduction : classification d'images
- Réseau de neurones (entrée, couche cachée, sortie)
- Évaluation de la classification
- Les convolutions 2D – Pooling - Softmax
- Architectures classiques des réseaux convolutionnels
- Apprentissage supervisé - Apprendre une « target function »
  - Loss/Rétro-propagation/Époques/Diversité/Batch normalization/Dropout
- Expérimenter avec un réseau
- Réutiliser un réseau (*Transfert Learning*)
- Conclusion

Introduction

Avènement de l'-**IA**- : définition du premier système à base de réseaux de neurones qui a dépassé l'état de l'art pour la classification d'image durant l'année…

Introduction

Résultats de la compétition ILSVRC) 2012
ImageNet Large Scale Visual Recognition Challenge (Classification)

'Supervision' : Approche neuronale : Krizhevsky et al. – erreur de **16.4%**

ISI : Second Meilleur (*dense SIFT*) – erreur de **26.2%**

Dataset ILSVRC :

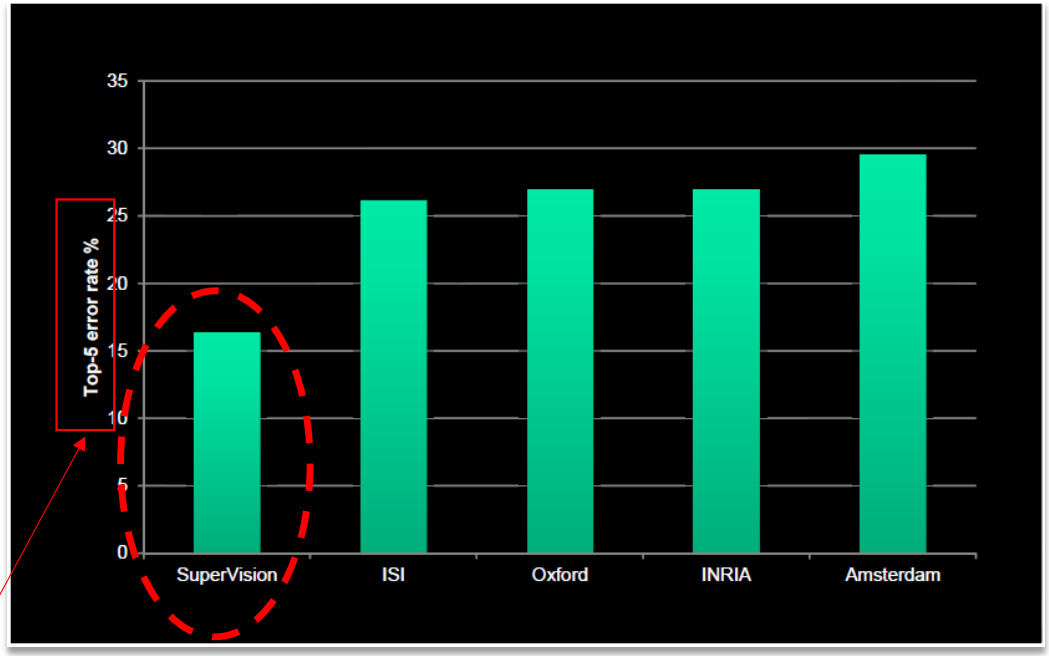
1000 catégories visuelles (*labels*)

150 000 images de test

Une image -> une et une seule catégorie "vraie"

Tâche : pour chaque image du test, trier les categories *prédites* de la plus à la moins probable

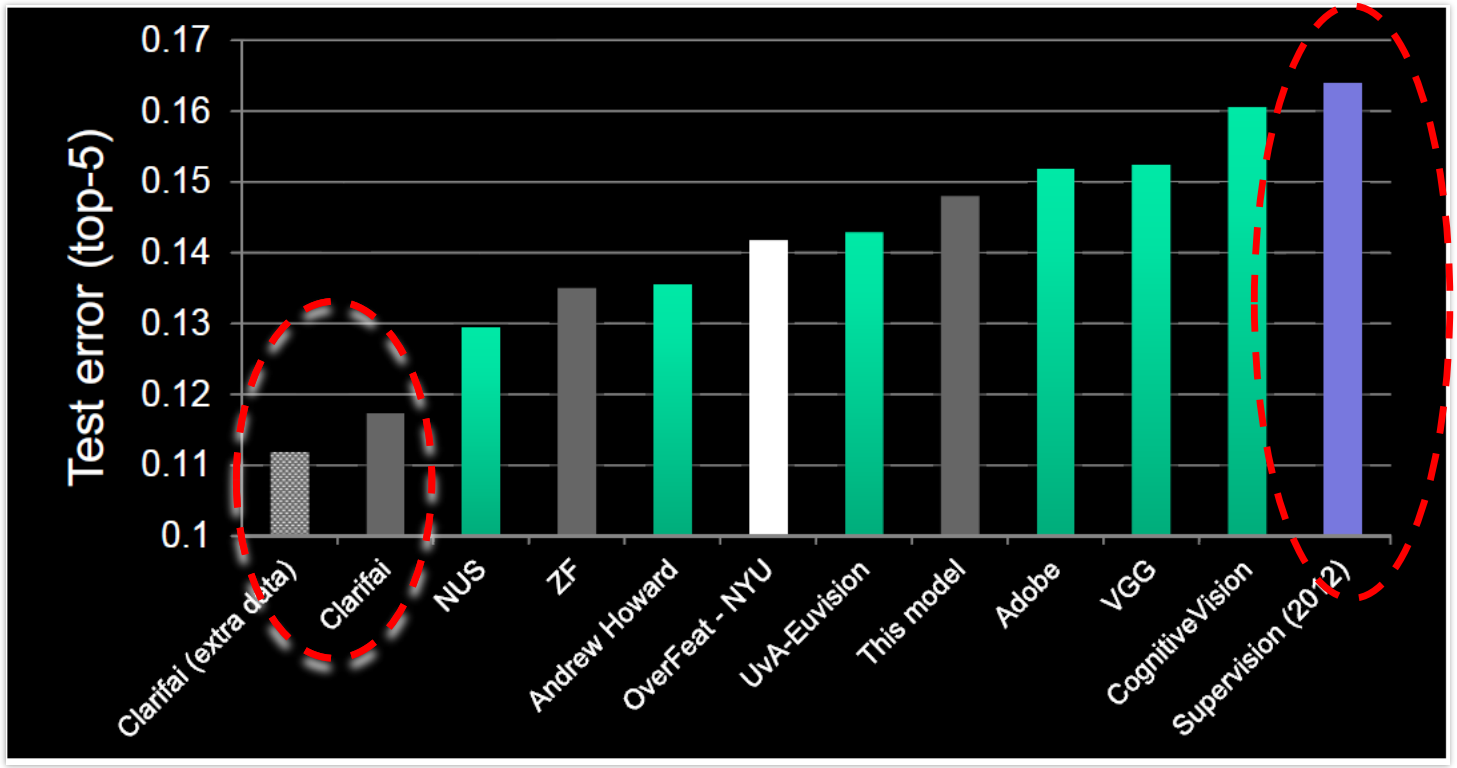
- Pour 1 image => un vecteur 1000D : prediction (nombre réél) pour chacune des 1000 categories



top-5 error rate (% des images pour lesquelles la catégorie "vraie" n'est pas dans les 5 prédites les plus probables)

Introduction
ImageNet Classification 2013 : amélioration supplémentaire

http://www.image-net.org/challenges/LSVRC/2013/results.php

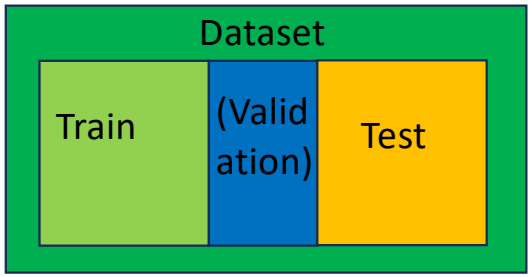


Introduction
Approche neuronale pour la classification

- Un réseau de neurones apprend (apprentissage supervisé)
  - Classification : chaque image a une (ou plusieurs) étiquette(s) (*label*)
  - On lui fournit en entrée de nombreux couples (entrée/label(s))
  - Il apprend les paramètres qui permettent de mieux prédire le lien entrée -> label(s)

- L'apprentissage, pour un réseau de neurones, a pour objectif de minimiser l'erreur entre
  - la prédiction (étiquette(s) prédite(s)) pour une image fournie: p(rocking chair) = 0.7
  - et la vérité-terrain (ex. : « vraie » étiquette) : p_{vrai}(rocking_chair) = 1
    - *Ex. : erreur* $|p_{vrai}(\text{rocking chair}) - p(\text{rocking chair})| = 0.3$

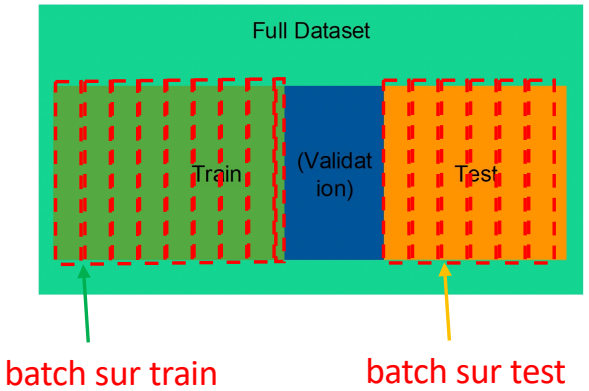
- Apprentissage sur un Dataset (collection)
  - *Ex. : ILSVRC*
    - *Train : 1,2M*
    - *Validation : 50K*
    - *Test : 150K*
    - *1000 labels*



Introduction
Approche neuronale pour la classification

- Utilisation d'un dataset sous forme de batchs (pour apprentissage et test)
  - Afin d'avoir des calculs très rapides de multiplications de matrices, les librairies de Deep Learning comme **`pytorch`** traitent plusieurs images à la fois, par des **batchs**
  - Découper un ensemble d'images **`E`** en batchs de taille **`n`** est simple (*résultat* **`Sbatch`** *:* un ensemble d'ensemble d'images), algo :

```
Contruit_batchs(entrées : E, n ; sortie Sbatch)
Etrav = E
Sbatch = {}
Tant que Etrav non-vide
    batch = tirage au hasard N images de Etrav
    Sbatch += {batch}
    Etrav -= batch
```



  - C'est typiquement ce que prépare un `data_loader` (cf. TP batchs de 128 images pour le train et 256 pour le test)

Introduction
Réseaux profonds (convolutionnels) / Deep Learning

Émergence due à une conjonction d'avancées de la recherche

- Travaux algorithmiques sur les réseaux de neurones
- Très grosses quantités de données annotées (ImageNet *actuel* : 14 M d'images, 22K labels)
- Puissance de calculs gigantesque des GPU (Nvidia Tesla V100 : 125 Teraflops (10^{12}), 300 Watts…)
  - Réseaux profonds possibles (>> 3 couches (cf. plus loin))
- Opérations spécifiques sur les images (convolutions)
- Fonctions d'activations efficaces : ReLU (Rectified Linear Unit)/sigmoïde
- Aides au processus d'apprentissage :
  - Batch normalization
  - Dropout
- Code utilisable assez facilement, des réseaux adaptables facilement

Introduction
Utiliser du Deep Learning est (+ ou -) facile

- **Maths**
  - algèbre linéaire
    - $Y = A.X + B$ (multidimentionnel avec des tenseurs)
  - calcul différentiel (utilisé durant l'apprentissage des réseaux)
    - $f(x+h) = f(x) + f'(x).h + o(h)$ (avec variables multidimensionelles)
    - $(g\ o\ f)'(x) = (g'o\ f)(x).f'(x)$ (appliqué récursivement)
- **Outils**: packages très bien intégrés, efficaces and faciles à utiliser
  - Python
  - Packages pour le calcul différentiel (*gradients*) : on peut s'intéresser uniquement à la partie algèbre linéaire

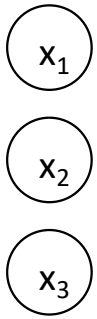
TPs : tous ces outils sont encapsulés pour permettre de faire des expérimentations "simplement"

Réseau de neurones

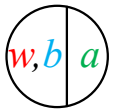
- Un réseau de neurones
  - contient des neurones.
  - est organisé en couches (une couche contient plusieurs neurones)

- Les couches de neurones
  - Les couches « cachées » : sans lien avec l'extérieur
  - Les couches d'entrée et de sortie qui interagissent avec l'extérieur
  - Elles ont toutes des comportements similaires.

Une couche « x » de 3 neurones :



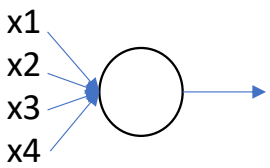
Réseau de neurones

Un neurone (de type *perceptron*) 

contient plusieurs éléments :

- Des entrées
- Une sortie

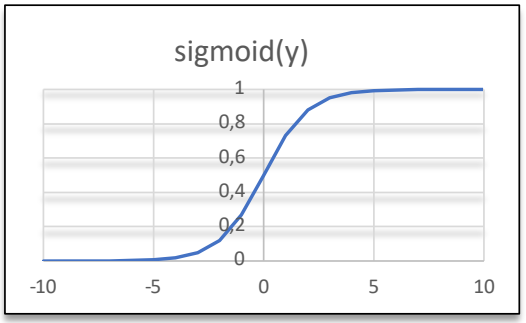
Exemple de neurone avec 4 entrées =>

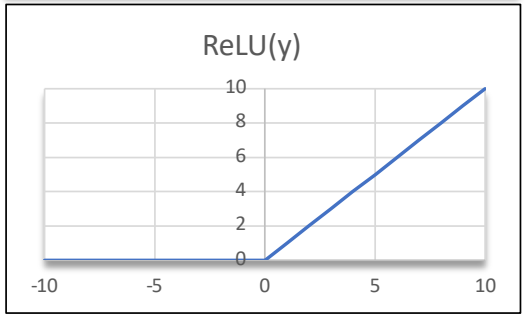


- Une partie « linéaire » qui pondère ses entrées, qui sort la valeur y
  - Les poids de connexion nombres réels, $w = <w_1, \ldots. w_n>$: vecteur avec autant de poids que d'entrées n
  - Un **biais** (facultatif), b, nombre réel
  - *Exemple* : si 4 entrées, $w = (w_1, w_2, w_3, w_4)$,

$$y = w_1.x_1 + w_2.x_2 + w_3.x_3 + w_4.x_4 + b$$

- Une partie possiblement non-linéaire a sur le réel y, *fonction d'activation*
  - *Linéaire* : $l(y) = y$
  - *sigmoid* (logistic), dans [0,1] : $\sigma(y) = \frac{1}{1+e^{-y}}$
  - *ReLU* : Rectified Linear Unit : $\mathrm{ReLU}(y) = \begin{cases} 0 \text{ si } y < 0 \\ y \text{ si } y \geq 0 \end{cases}$
  - *tanh* : tangente hyperbolique, dans [-1, 1] : $tanh(y) = \frac{e^{y}-e^{-y}}{e^{y}+e^{-y}}$
- On note out_n la valeur de sortie d'un neurone n, $out_n = a(y)$

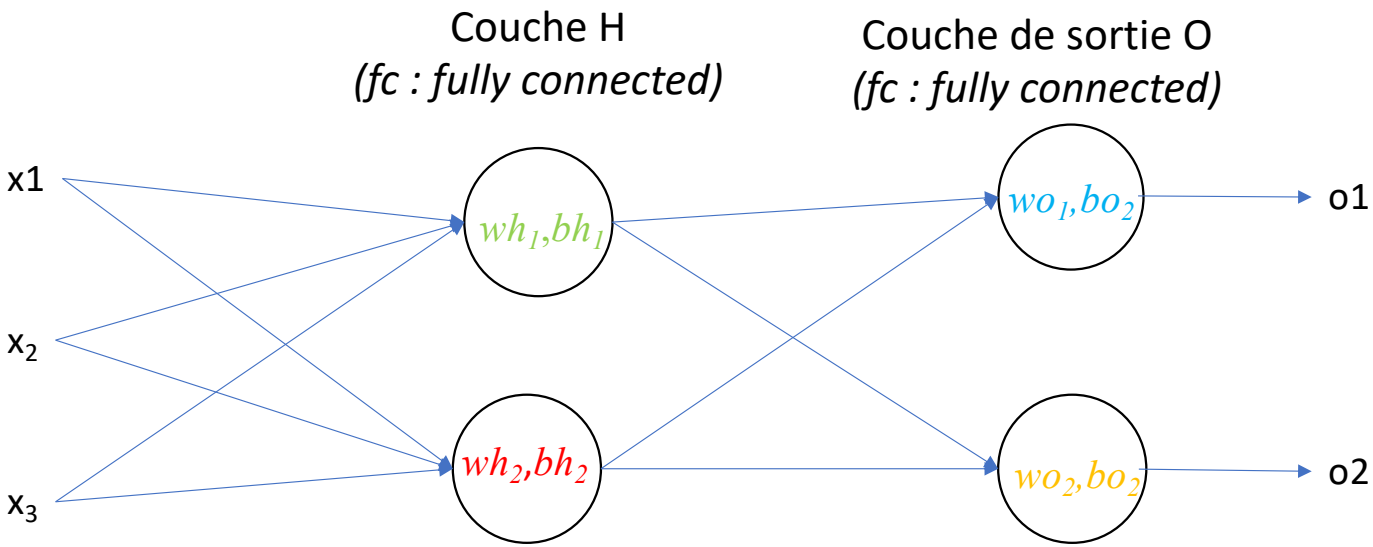


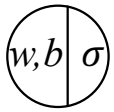


https://developers.google.com/machine-learning/crash-course/neural-networks/nodes-hidden-layers?hl=fr

Réseau de neurones (perceptron multicouche)

Architecture d'un réseau de neurones R simple :
2 couches = 1 d'entrée H (2 neurones), 1 de sortie O (2neurones), fonction d'activation σ pour tous



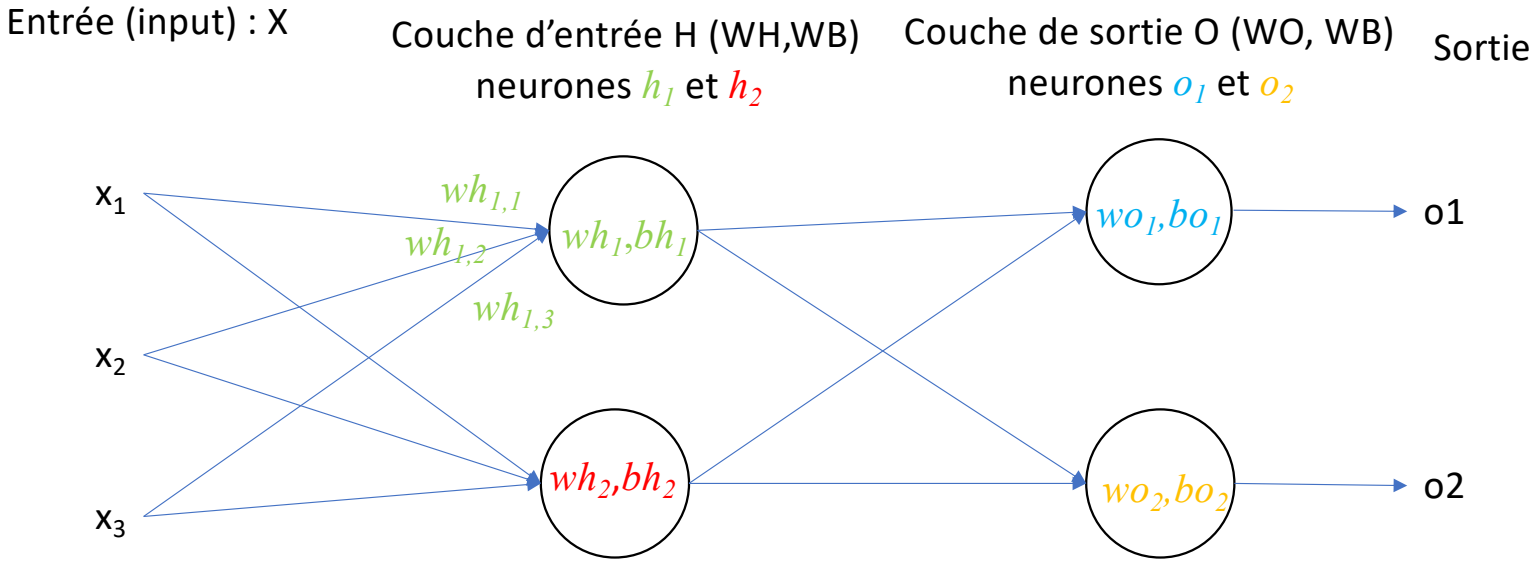
Ecriture simplifiée  des neurones avec une fonction d'activation σ,  .

Nombre de paramètres : couche H : 3 * 2 + 2 = 8, couche O : 2 * 2 + 2 = 6, donc **14 paramètres** à apprendre
Note : ne pas confondre **paramètre** et *hyper-paramètre* (non-appris) : nombre de couches, learning rate, … (voir plus loin)

Réseau de neurones (perceptron multicouche)

Inférence (forward pass) sur R

- le réseau (qui a tous ses poids fixés) prend les valeurs en entrée, et calcule les valeurs en sortie



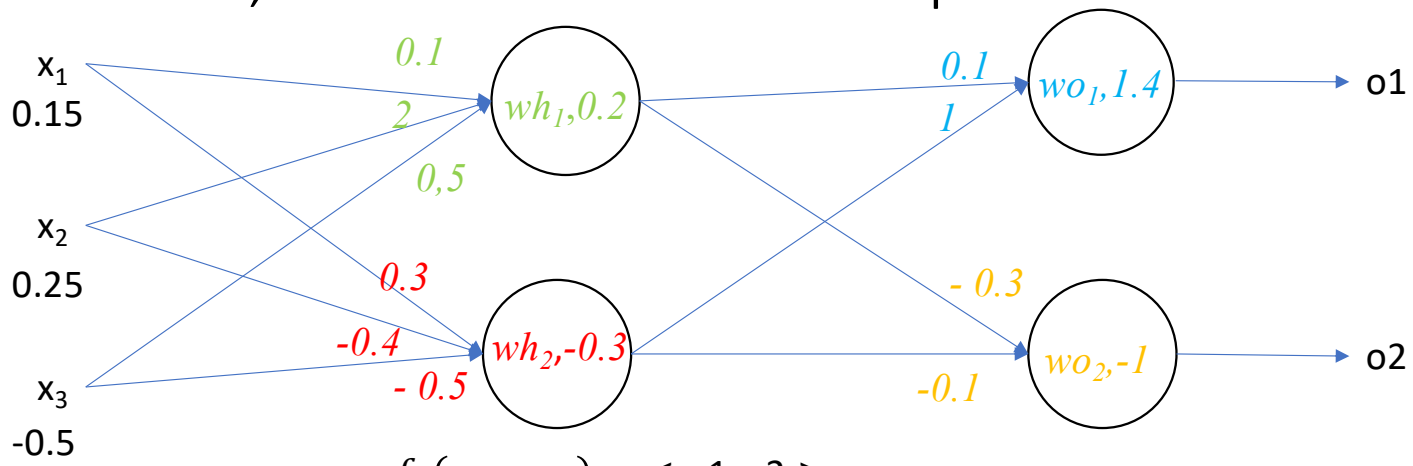
Rappel : $wh_1=<wh_{1,1}, wh_{1,2}, wh_{1,3}>$, ...

WH=<*wh1*,*wh2*> (…c'est un tenseur… vocabulaire classique des librairies de réseaux de neurones)

$$o1 = out_{o1} = f_R(x_1, x_2, x_3) = \sigma\left(wo_{1,1} * out_{h1} + wo_{1,2} * out_{h2} + bo1 \right)$$

Réseau de neurones (perceptron multicouche)

Reprenons notre réseau, cette fois avec des valeurs de poids



$f_R(x_1, x_2, x_3) = < o1, o2 >$

$$out_{o1} = f_R(x_1, x_2, x_3) = \sigma\left(wo_{1,1} * out_{h1} + wo_{1,2} * out_{h2} + bo1 \right)$$

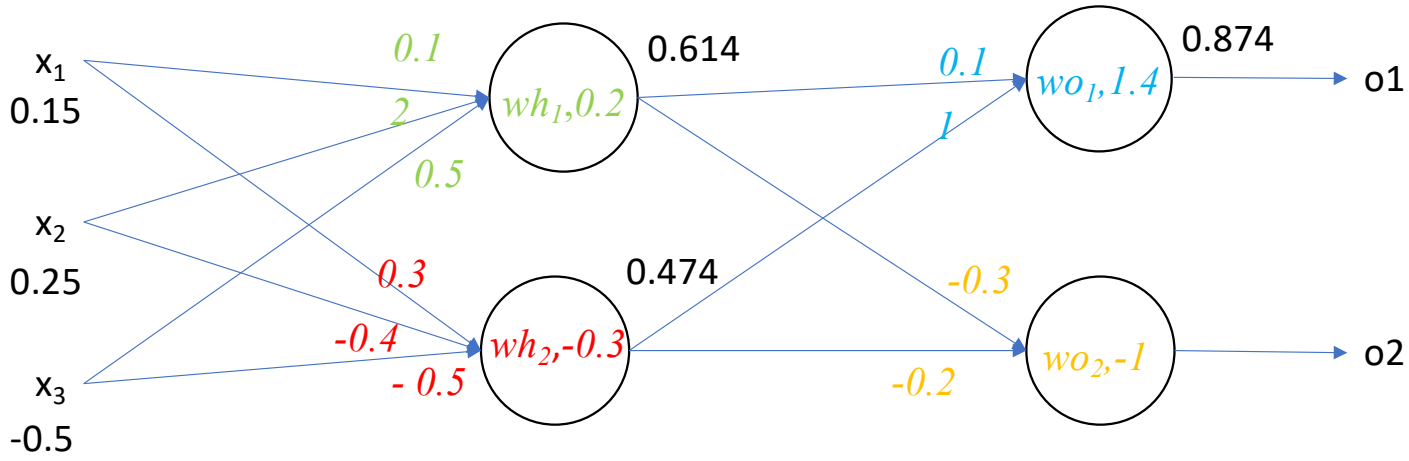
$$= \sigma \begin{pmatrix} wo_{1,1} * \sigma\left(wh_{1,1} * x_1 + wh_{1,2} * x_2 + wh_{1,3} * x_3 + bh_1\right) \\ + \; wo_{1,2} * \sigma\left(wh_{2,1} * x_1 + wh_{2,2} * x_2 + wh_{2,3} * x_3 + bh_2\right) \\ + \; bo_1 \end{pmatrix}$$

$$out_{o2} = f_R(x_1, x_2, x_3) = \sigma \begin{pmatrix} wo_{2,1} * \sigma\left(wh_{1,1} * x_1 + wh_{1,2} * x_2 + wh_{1,3} * x_3 + bh_1\right) \\ + \; wo_{2,2} * \sigma\left(wh_{2,1} * x_1 + wh_{2,2} * x_2 + wh_{2,3} * x_3 + bh_2\right) \\ + \; bo_2 \end{pmatrix}$$

Réseau de neurones (perceptron multicouche)

Reprenons notre réseau, cette fois avec des valeurs de poids

Rappel :
$$\sigma(y) = \frac{1}{1+e^{-y}}$$



$$\text{out}_{o1} = f_R(x_1, x_2, x_3)$$

$$= \sigma\begin{pmatrix} wo_{1,1} * \sigma(wh_{1,1} * x_1 + wh_{1,2} * x_2 + wh_{1,3} * x_3 + bh_1) \\ + \ wo_{1,2} * \sigma(wh_{2,1} * x_1 + wh_{2,2} * x_2 + wh_{2,3} * x_3 + bh_2) \\ + \ bo_1 \end{pmatrix}$$

$$= \sigma\begin{pmatrix} 0.1 * \sigma(0.1 * x_1 + 2 * x_2 + 0.5 * x_3 + 0.2) \\ + 1 * \sigma(0.3 * x_1 - 0.4 * x_2 - 0.5 * x_3 - 0.3) \\ + \ 1.4 \end{pmatrix}$$

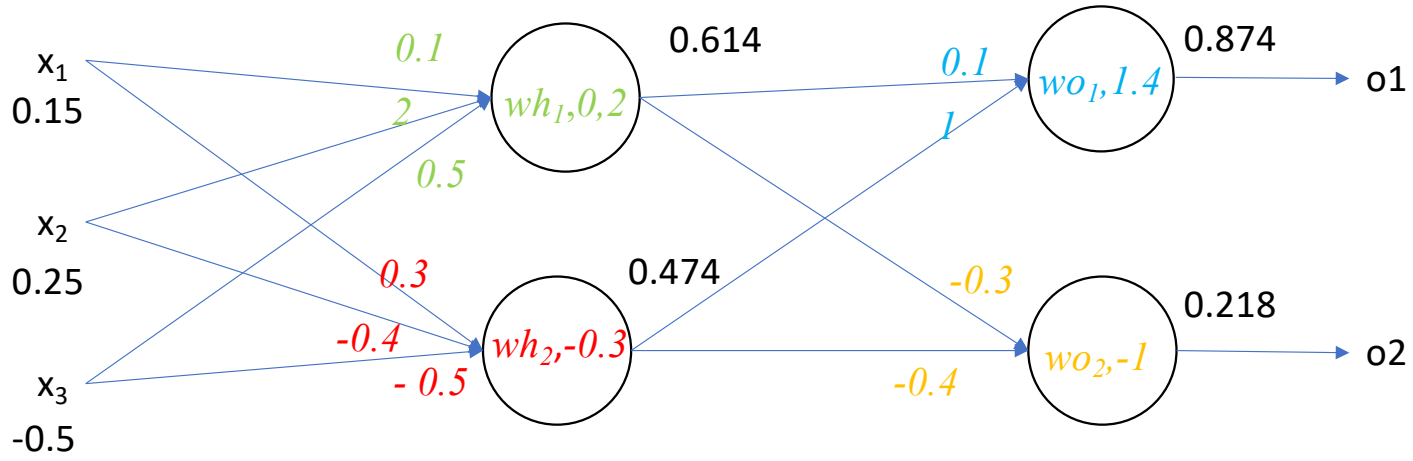
$$\text{out}_{o1} = f_R(0.15, 0.25, -0.5) = \sigma\begin{pmatrix} 0.1 * \sigma(0.465) \\ +1 * \sigma(-0.105) \\ + 1.4 \end{pmatrix}$$

$$= \sigma\begin{pmatrix} 0.1 * 0.614 \\ +1 * 0.474 \\ + 1.4 \end{pmatrix}$$

= 0.874

Réseau de neurone (perceptron multicouche)

Reprenons notre réseau, cette fois avec des valeurs de poids



o2 = 0.218 <= Exercice : faire le calcul

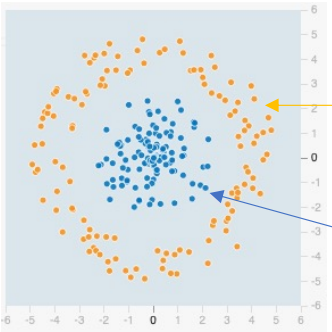
Réseau de neurone (perceptron multicouche)

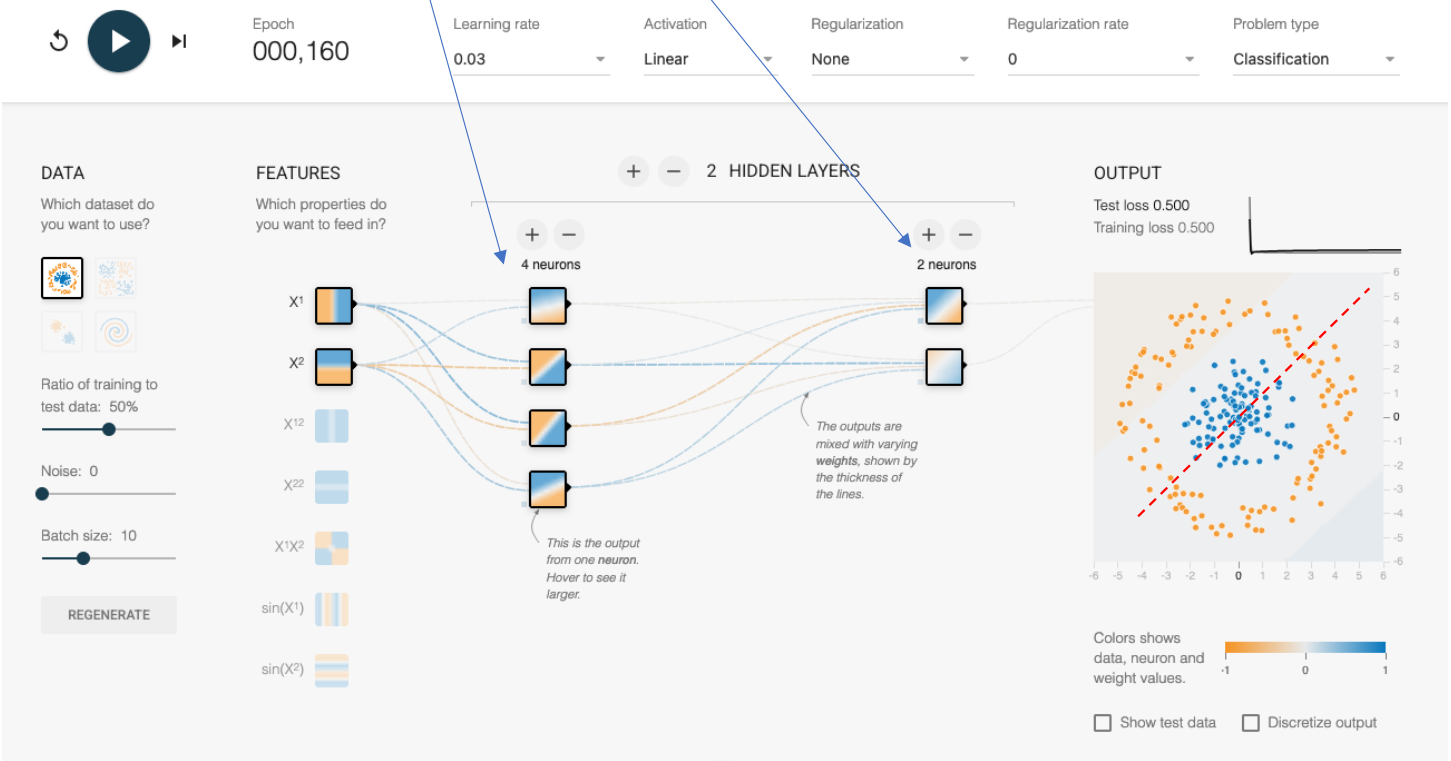
Pour déclarer ce réseau de **2 couches de 2 neurones** avec PyTorch en python :

```
import torch
import torch.nn as nn

# Définition du modèle
class MyNet(nn.Module):  #definition d'une classe qui décrit des réseaux voulus
      def __init__(self):
            super(MyNet, self).__init__()
            self.fc1 = nn.Sequential( # Première couche : H
                nn.Linear(3, 2),  # Partie linéaire : 3 entrées, 2 neurones
                nn.Sigmoid())     # Activation : Sigmoïde pour chaque neurone de la couche
            self.fc2 = nn.Sequential(# Deuxième couche : O
                nn.Linear(2, 2),  # Partie linéaire: 2 entrées, 2 neurones
                nn.Sigmoid())     # Activation Sigmoïde pour chaque neurone de la couche

      def forward(self, x): # description de l'enchaînement des couches à l'inférence
            x = self.fc1(x) # d'abord la couche O appliquée à l'entrée
            x = self.fc2(x) # ensuite la couche H
            return x

modele = MyNet() # Crée une instance de MyNet avec des poids aléatoires = un réseau

# Exemple d'inférence avec 3 caractéristiques
exemple_entrée = torch.randn(1, 3)  # 1 échantillon d'entrée aléatoire (x1,x2,x3)
sortie = modele(exemple_entrée) # inférence avec entrée fournie
print(sortie) # Affiche la sortie
```

Réseau de neurones

- Pourquoi des fonctions d'activation non-linéaires
  - Cas de données non-séparables linéairement :
    - 2 couches (une de 4 + une de 2 neurones)
    - fonction d'activation linéaire ($f(y) = y$)



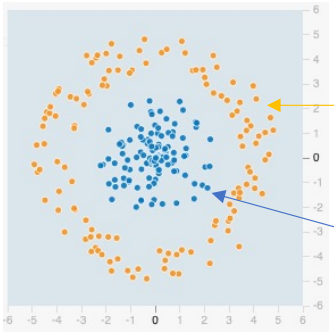


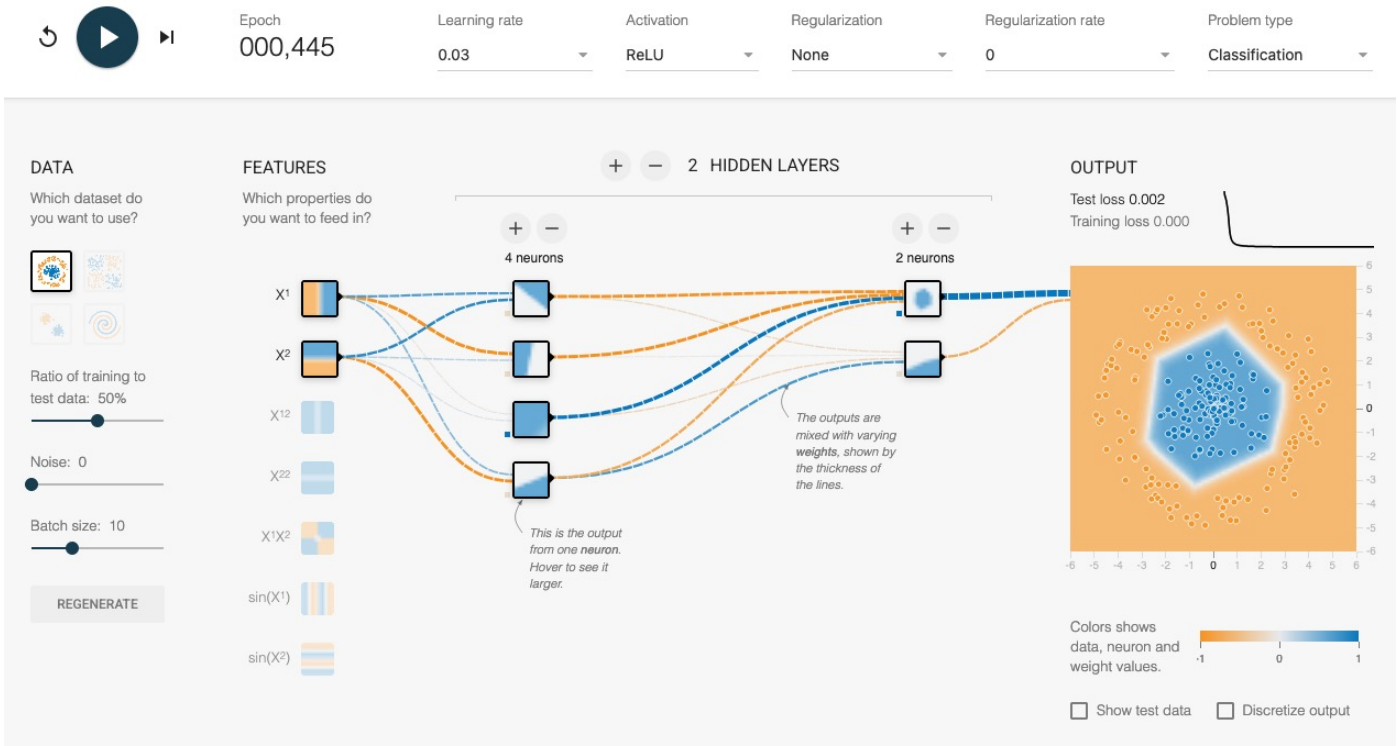
Séparation non-apprise 50% d'erreur

https://playground.tensorflow.org/

Réseau de neurones

- Pourquoi des fonctions d'activation non-linéaires
  - Cas de données non-séparables linéairement :
    - 2 couches (une de 4 + une de 2 neurones)
    - fonction d'activation ReLU





Séparation apprise
0.2% d'erreur

https://playground.tensorflow.org/

Evaluation de la classification (*accuracy*)

- Sur un ensemble de test T = {(image, vrai label)}
  - Exemple : 2 labels {chien, chat}
    - On teste un réseau avec deux sorties : o1=*p*(chien), o2=*p*(chat)

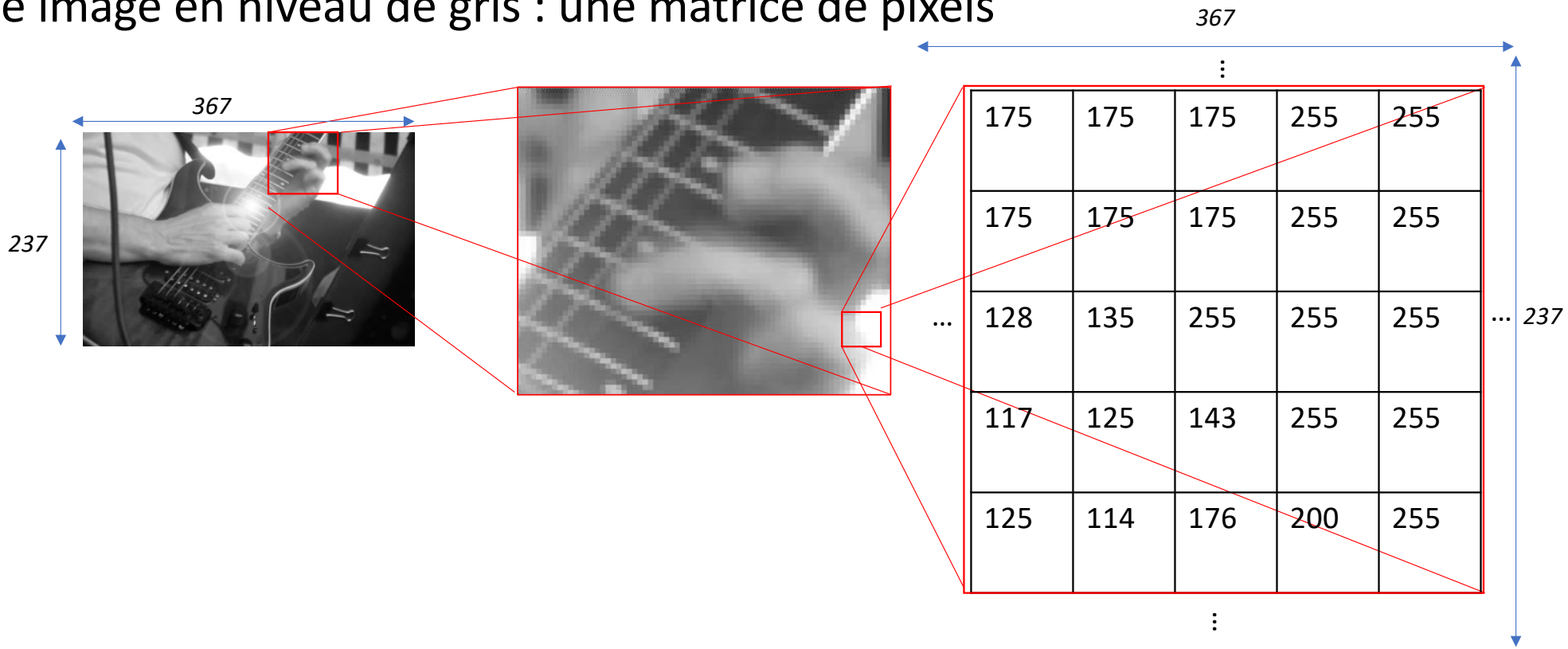
    - Pour une image *i*
      - Si la valeur *p* la plus élevée de o1 et o2 correspond au vrai label,
        alors « accuracy » = 1 pour *i*
        sinon « accuracy » = 0 pour *i*

        (équivalent à la précision@1 document)

  - Pour l'ensemble de test T
    - acc. = la moyenne des valeurs « accuracy » des images de T
    - Équivalent à P@1

Les convolution 2D
Élements spécifiques pour les images

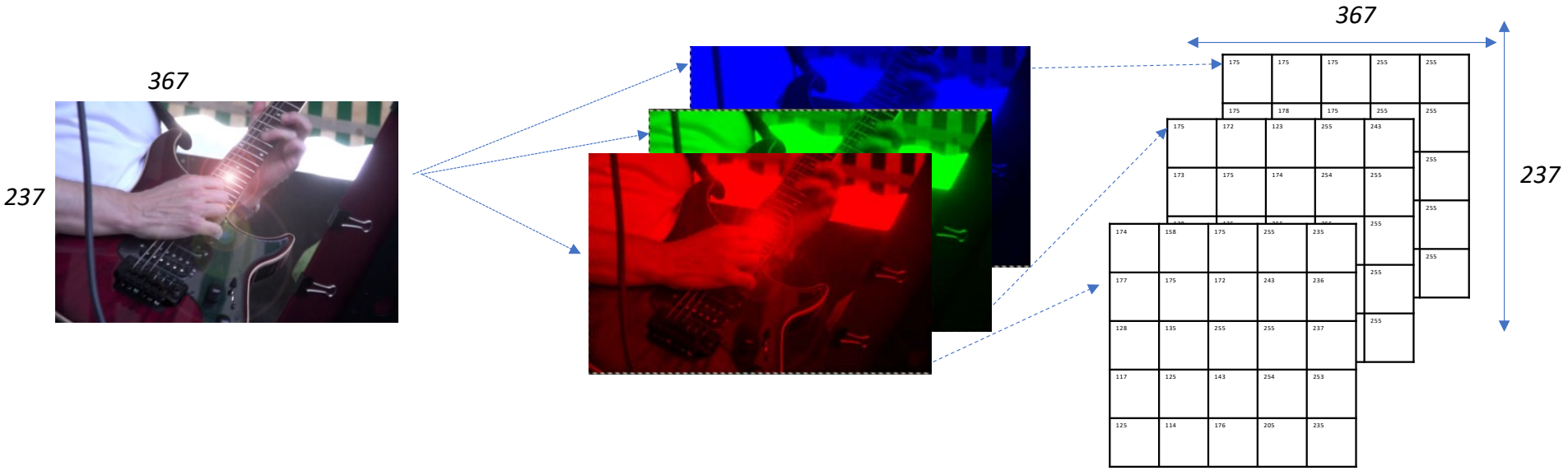
- Une image en niveau de gris : une matrice de pixels



Les convolutions 2D
Élements spécifiques pour les images

- Une image en couleur : 3 canaux (= (1 Red, 1 Green, 1 Blue)

=> 3 matrices de pixels



- Image (367x237x3) = 260 937 éléments (pixels)

Les convolutions 2D - Pourquoi

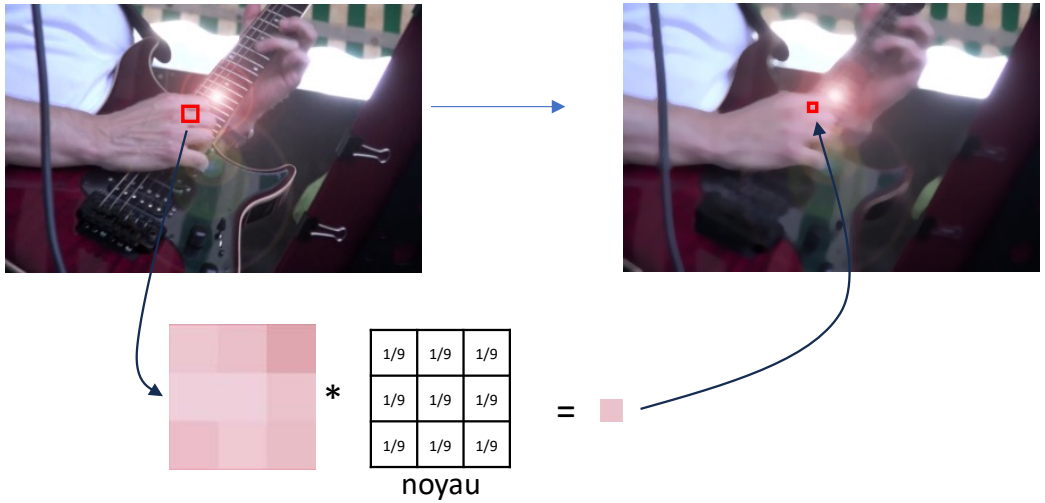
- On peut théoriquement utiliser les perceptrons sur des images :
  - Il faut une entrée par pixel (de chaque canal)
    - Dans notre exemple, image en noir et blanc (1 canal) avec 260 937 (= 367 * 237) pixels
      - Pour une couche H de 30 neurones, on a 260 937 * 30 + 30 paramètres = 7 828 140 paramètres
  - Plus on a de paramètres à apprendre, plus il faut d'exemples d'entrainement pour pouvoir apprendre les paramètres avec assez de qualité…
  - Une couche de perceptron n'utilise pas les voisinages entre pixels

  ⇒ Les perceptrons multicouches ne sont pas adaptés aux images

- Besoin de couches mieux adaptées aux images, avec moins de paramètres, capables d'intégrer des voisinages de pixels
  - les convolutions 2D !

Les convolutions 2D – pré-existant

- Les convolutions 2D sont le fruit d'années de recherche en traitement d'images, et particulièrement les filtres
  - Un filtre applique itérativement le même noyau de taille w x h à une image



noyau

Flou médian de taille 3 x 3

  - Un filtre de convolution a peu de paramètres et utilise le voisinage de pixels

Les convolutions 2D - pré-existant

- Exemple animé, filtre (3x3) de noyau

1	-1	1
-1	0	-1
1	-1	1

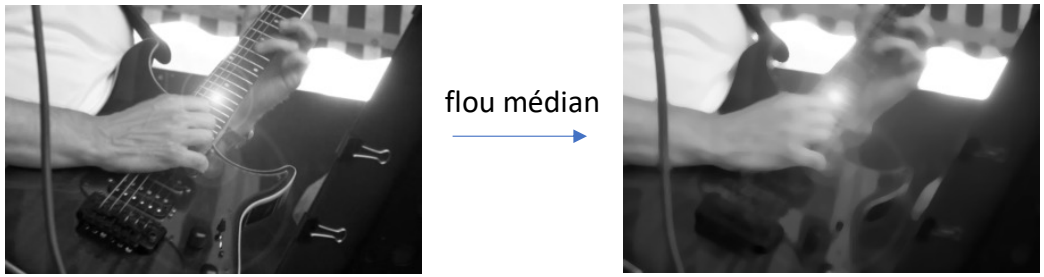
qui détecte des coins sur un canal (niveau de gris)

- …. ici le filtre a seulement quelques paramètres prédéfinis (9 pour un noyau de 3x3) et est adapté à une détection

https://medium.com/@BriceNicodem/convolutional-neural-networks-theoretical-aspects-fec80715a483

Les convolutions 2D - les channels

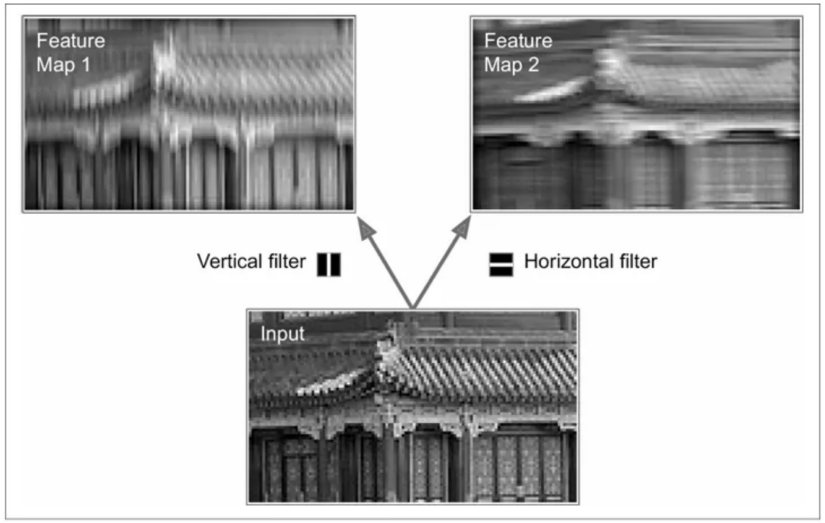
- Notes importante
  - Le résultat de l'application d'un filtre est UNE IMAGE, aussi appelée **canal** (ou *channel*) dans la suite



  - La taille de l'image en sortie est, par défaut, plus petite que celle en entrée

Les convolutions 2D

- L'idée des convolution 2D dans le Deep Learning est d'apprendre les poids d'un filtre, qui sont optimisés pour une tâche donnée
- Un filtre de convolution appris va détecter une caractéristique
  - il faut beaucoup de filtres pour détecter beaucoup de choses différentes
  - *Ex. 2 filtres sur une image mono-canal* :



Reference: Geron, A. (2019). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd ed.). O'Reilly Media.

Les convolutions 2D

- Des filtres, avec paramètres appris, pour extraire des informations locales utiles pour la classification d'images
- Principe
  - Petits filtres (Largeur X Hauteur X Profondeur, ex. 3 pixels * 3 pixels * 1 canal)
  - Glisser sur la largeur et la hauteur du *volume* d'entrée (Largeur X Hauteur X Profondeur)
    - Profondeur = nombre de canaux
- Avantages
  - Prise en compte d'éléments spatiaux locaux
  - Filtres apprenables, peu de paramètres
  - Invariance spatiale / de translation
  - Poids partagés
  - Le même filtre est appliqué sur un canal de toute l'image

Les convolutions 2D

- Les paramètres appris d'un convolution2D
  - Les poids des coefficients du noyau de la convolution + un biais éventuel
    - Ex. 1 canal, convolution 3x3 => 9 paramètres, + 1 biais : 10 paramètres
      - Exemple de filtre appris avec le biais

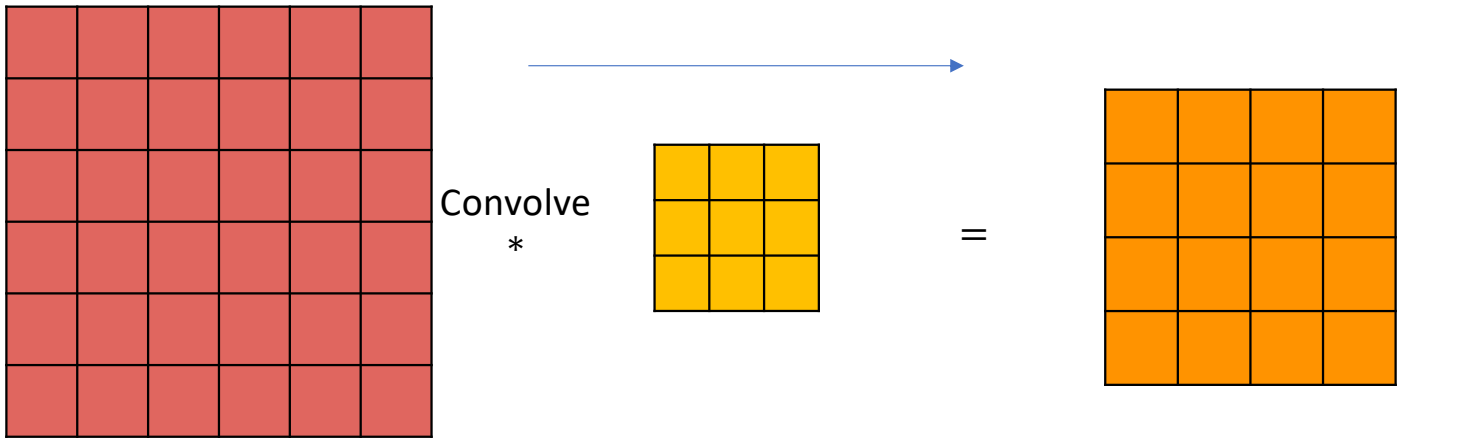
-0.3	0.6	0.1
-1.2	0.2	-1.1
0.3	0	1.4

0.1

- Les (hyper-)paramètres fixés d'une convolution 2D
  - Stride : le pas de déplacement entre l'application de 2 convolutions (défaut = 1)
  - Padding : l'ajout de bord au canal d'entrée (défaut = 0)

Les convolutions 2D

- Pour un canal (= depth), par exemple un niveau de gris (1 channel)



note : biais non indiqué

(Paramètres: Stride = 1, Padding = Valid, Kernel size = 3 x 3 x 1, Number of filters = 1)

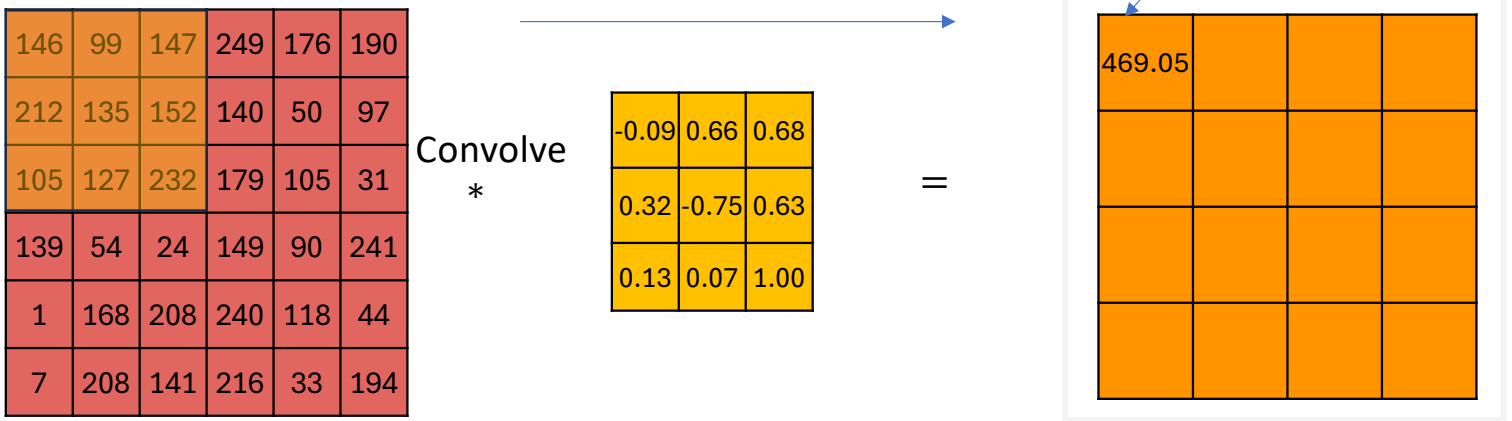
Déclarer cette convolution avec PyTorch : `nn.Conv2d(1, 1, kernel_size=3)`

Rappel, par défaut : padding = 0, stride =1, et biais actif

Les convolutions 2D

= (146*-0.09) + (99*0.66) + (147*0.68) +(212*0.32) + (135*-0.75)
+ (152*0.63) + (105*0.13) + (127*0.07) + (232*1.00)

avec biais = 0



Input volume
(height x width x depth)
6 x 6 x 1

Filters
(height x width x depth)
3 x 3 x 1

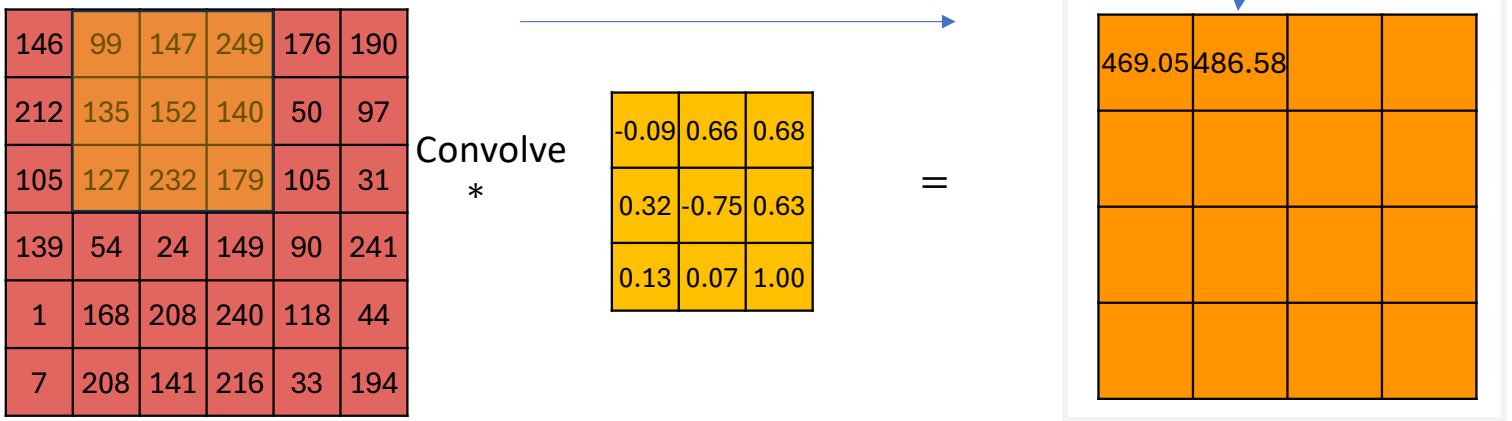
Result – feature map
(height x width x depth)
4 x 4

(CNN parameters: Stride = 1, Padding = 0, Kernel size = 3 x 3 x 1, Number of filters = 1)

Les convolutions 2D

= (99*-0.09) + (147*0.66) + (249*0.68) +(135*0.32) + (152*-0.75)
+ (140*0.63) + (127*0.13) + (232*0.07) + (179*1.00)

avec biais = 0



Input volume
(height x width x depth)
6 x 6 x 1

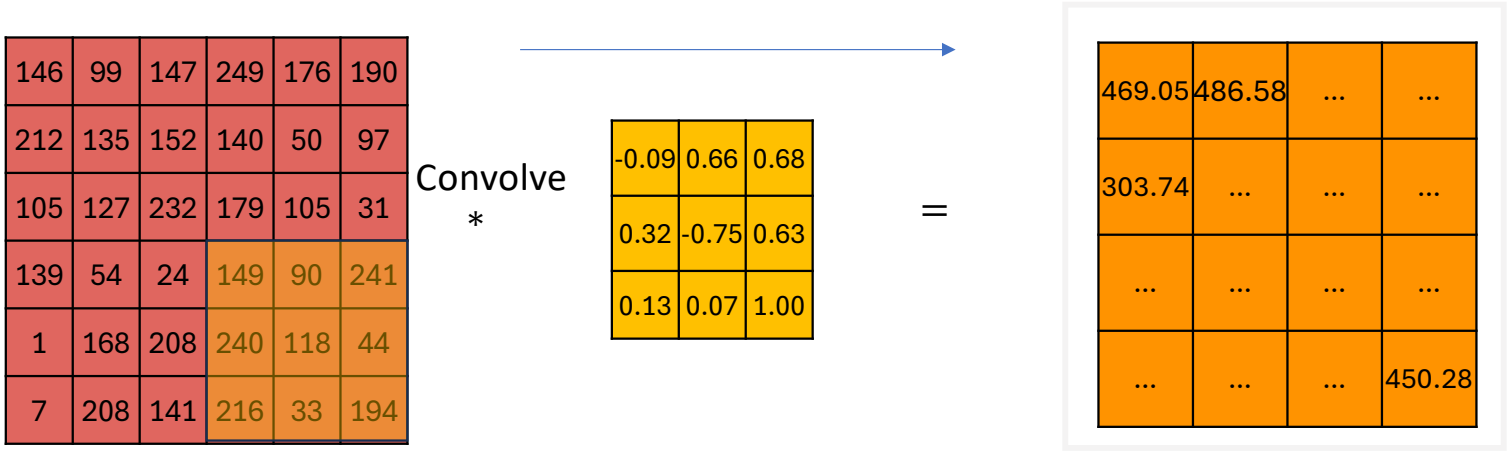
Filters
(height x width x depth)
3 x 3 x 1

Result – feature map
(height x width x depth)
4 x 4

(CNN parameters: Stride = 1, Padding = 0, Kernel size = 3 x 3 x 1, Number of filters = 1)

Les convolutions 2D

avec biais = 0



Input volume
(height x width x depth)
6 x 6 x 1

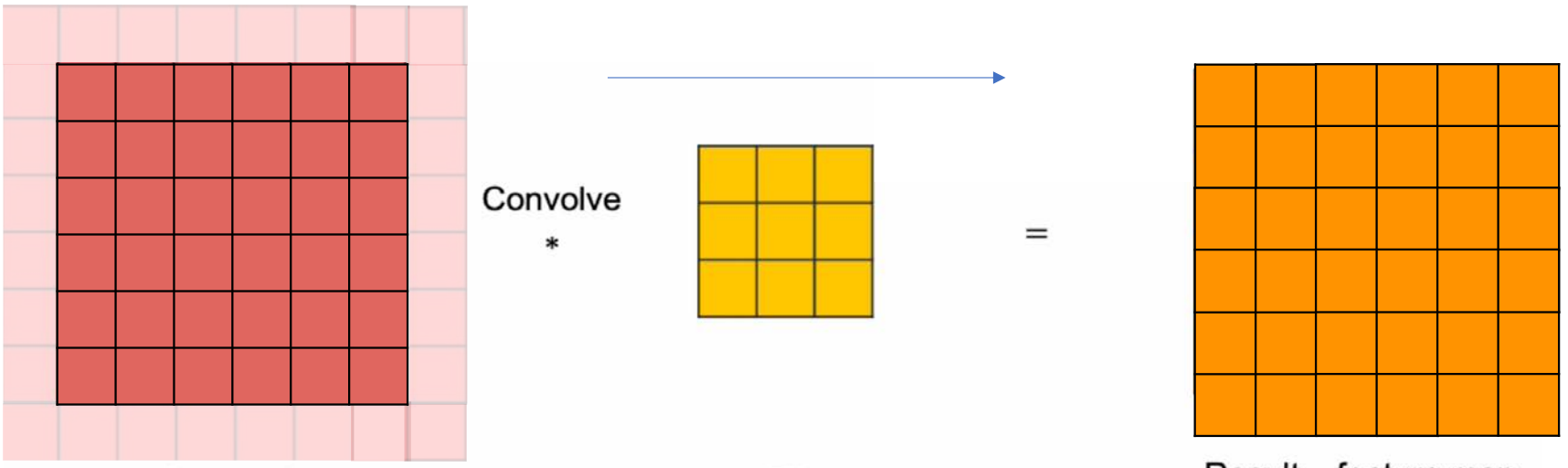
Filters
(height x width x depth)
3 x 3 x 1

Result – feature map
(height x width x depth)
4 x 4

(CNN parameters: Stride = 1, Padding = 0, Kernel size = 3 x 3 x 1, Number of filters = 1)

Les convolutions 2D

- Avec `padding=0`, l'application d'une convolution diminue la taille de la sortie
- Le padding ajoute des « bords » à l'entrée avant d'appliquer la convolution
  - Ex. avec padding de 1 : l'image d'entrée (6x6) devient alors de (6+1) x (6+1)

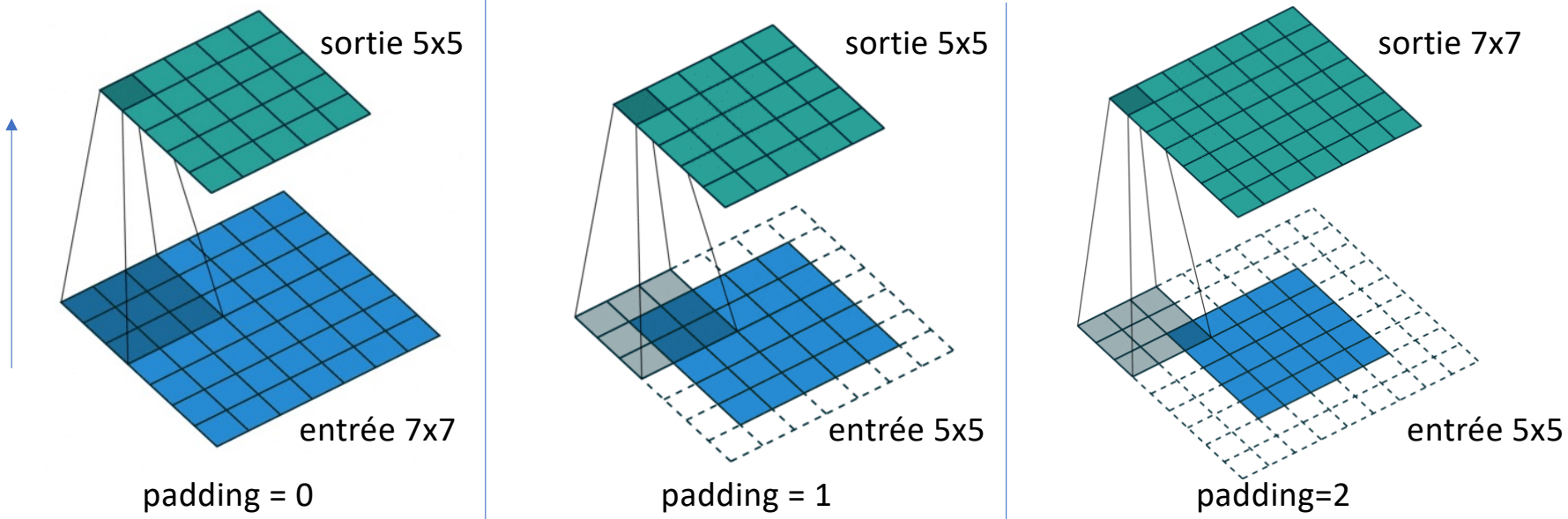


(CNN parameters: Stride = 1, Padding = 1 , Kernel size = 3 x 3 x 1, Number of filters = 1)

```
nn.Conv2d(1, 1, kernel_size=3, padding=1)
```

Les convolutions 2D

3 exemples de padding, avec stride=1,, noyau de 3x3



https://deeplizard.com/learn/video/YRhxdVk_sIs

Les convolutions 2D

- Application d'une convolution 2D sur un channel
  - Taille de filtre : k
  - Stride : s
  - Padding : p
  - Pour une image H_{in} x W_{in}

La taille de sortie (H_{out}, W_{out}) est :

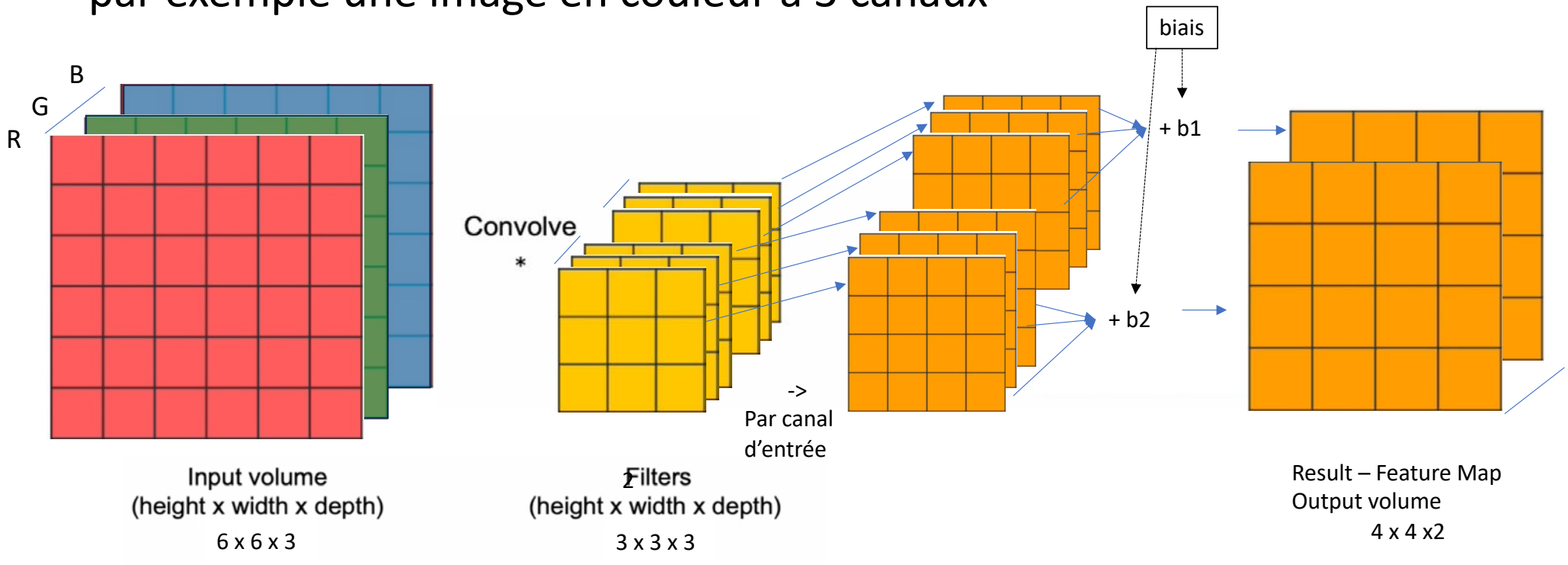
$$H_{out} = \frac{H_{in} - k + 2*p}{s} + 1$$

$$W_{out} = \frac{W_{in} - k + 2*p}{s} + 1$$

- Classiquement une image est carrée $H_{in} = W_{in}$
- Ex. : une convolution 3x3 avec padding=0, stride=1, sur une image 6x6 donne une image 4x4
- Si on a plusieurs channel la taille est même pour tous les channels

Les convolutions 2D

- Les convolutions peuvent manipuler des entrées à plusieurs canaux, par exemple une image en couleur à 3 canaux

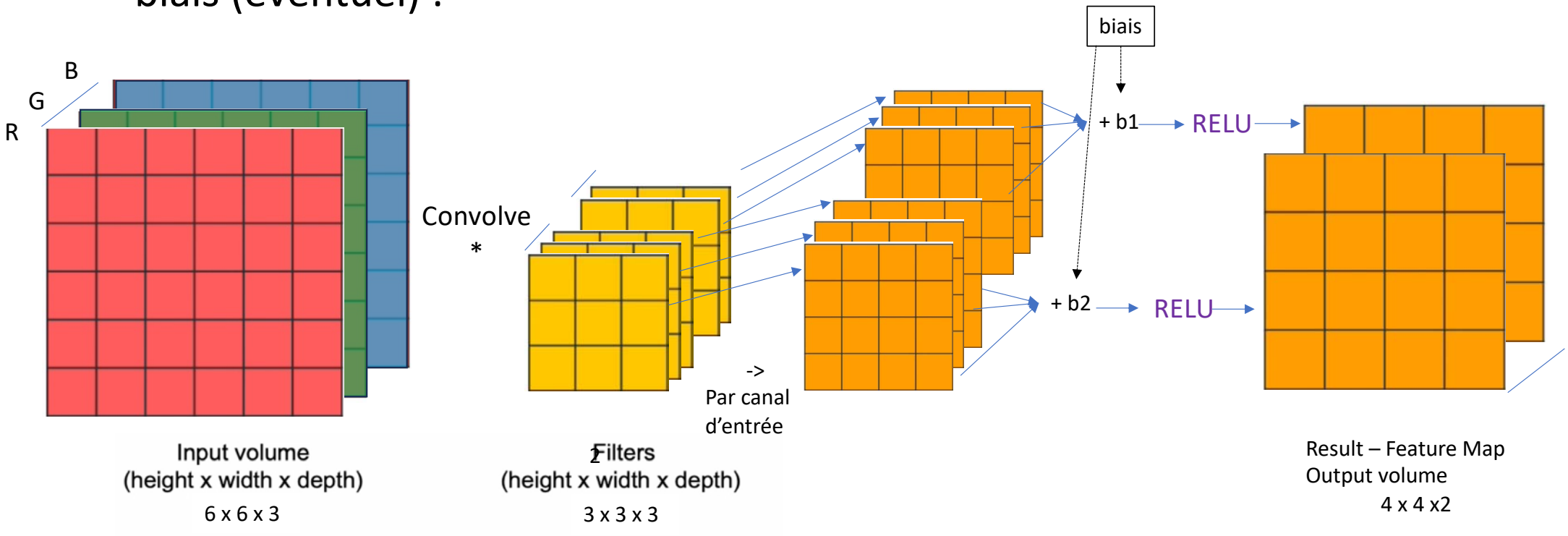


Cette convolution avec 3 canaux en entrée et 2 canaux en sortie avec pytorch avec biais :

```
nn.Conv2d(3,2,kernel_size=3)
```

Les convolutions 2D (NEW) avec activation

- Si activation RELU, alors cette activation s'applique après l'ajout du biais (éventuel) :



Les convolutions 2D

- Exemple tiré des TPs, convolutions2D avec activation ReLU :

```
self.layer1 = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3),
            nn.ReLU())
self.layer2_1 = nn.Sequential(
            nn.Conv2d(16, 16, kernel_size=3),
            nn.ReLU())
```

  - Si on a, en entrée, une image 28x28x3
    - Quelle est la taille H x W x D de la sortie de la couche layer1, de la couche layer2 ?
    - Combien de paramètres (appris) pour la couche layer1, pour la couche layer2_1, en considérant que les convolutions ont des biais ?

Les couches de pooling

Traitements de channels qui font des opérations prédéfinies sur des canaux, sans apprentissage

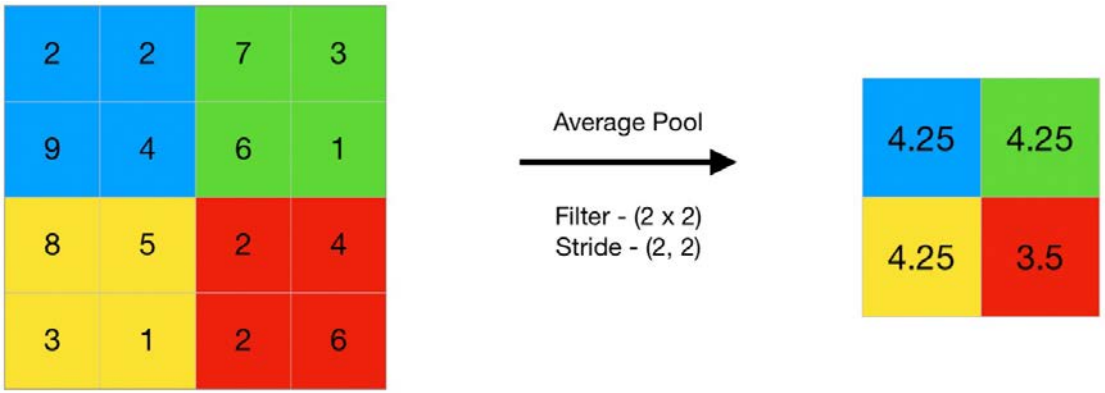
Objectif :

=> synthétiser l'information en réduisant les channels

https://medium.com/@danushidk507/max-pooling-ef545993b6e4

Les couches de pooling

- AvgPooling2D
  - Calcule la valeur moyenne d'une zone par channel, ex. 2x2



⇒ Ce traitement est strictement le même qu'une convolution 2D (¼, ¼, ¼, ¼)

⇒ MAIS <u>aucun paramètre appris (pas de biais)</u>

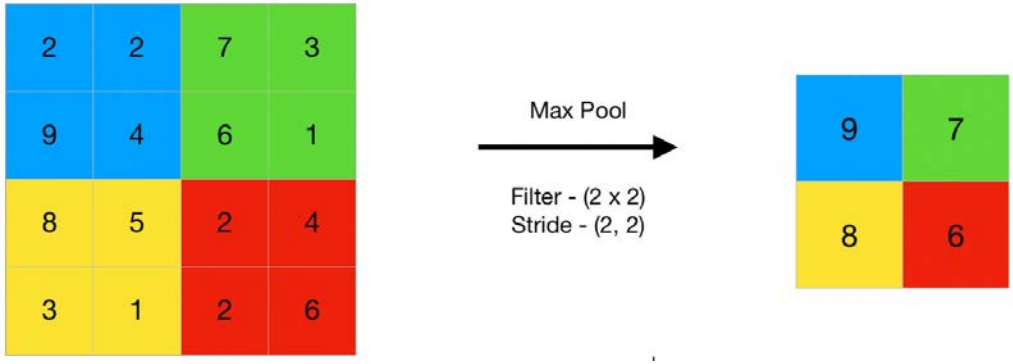
  - Classiquement pas d'activation après ces couches de Pooling
  - Nombre de channels en sortie = nb de channels en entrée

Déclaration de cette convolution avec pytorch : `nn.AvgPool2d(kernel_size=2, stride=2)`

https://medium.com/@danushidk507/max-pooling-ef545993b6e4

Les couches de pooling

- MaxPool2D
  - Calcule la valeur max d'une zone par channel, ex. 2x2



⇒ traitements « similaires » à une convolution, mais avec calcul de Max

⇒ <u>Aucun paramètre appris (pas de biais)</u>

  - Classiquement pas d'activation après ces couches de Pooling
  - Nombre de channels en sortie = nb de channels en entrée

Déclarer cette convolution avec pytorch : `nn.MaxPool2d(kernel_size=2, stride=2)`

https://medium.com/@danushidk507/max-pooling-ef545993b6e4

Les couches de pooling

Traitement de channels avec des traitements qui font des opérations prédéfinies sur des zones

- Hyperparamètres (similaires aux Convolutions 2D)
  - Taille du « noyau »
  - Padding
  - Stride
- Aucun paramètre appris
- Classiquement pas d'activation après ces couches de pooling
- Nombre de channels en sortie = nb de channels en entrée

https://medium.com/@danushidk507/max-pooling-ef545993b6e4

Les convolutions 2D + Pooling

- Exemple tiré des TPs, avec 2 couches de Conv2D suivies d'un MaxPool2D :

```
self.layer1 = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=3),
    nn.ReLU())

self.layer2 = nn.Sequential (
     nn.Conv2d(16, 16, kernel_size=3),
     nn.ReLU())

self.layer3 = nn.Sequential(
            nn.MaxPool2d(kernel_size=2, stride=2))
```

  - Si on a en entrée du réseau une image 28x28x3
    - Quelle est la taille de la sortie de la couche layer3 (channels, taille des channels) ?
    - Combien de paramètres (appris) pour la couche layer3 ?

Autre couche importante

- Le softmax
  - Appliqué sur valeurs réelles $< x_1, x_2, x_3 , \dots , x_n >$
  - Le softmax d'une de ces valeurs, ex. x_1, par rapport à l'ensemble de valeurs est :

$$s_x(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

  - Calcule des valeurs de *probabilités*, et sépare la valeur la plus grande des autres
  - Exemple : le softmax de < -1, 0, 3, 5 > donne < 0.002, 0,006, 0.118, 0.874 >
- Utilisé classiquement à la sortie d'un réseau dans la classification multi-classes
  - appliqué sur un vecteur de valeurs (en entrée) et sort le softmax sur la sortie : la dimension de l'entrée = dimension de la sortie

- Dans les TPs, utilisé hors réseau (une spécificité de l'entropie croisée de PyTorch, voir plus loin…) sur les sorties d'un batch :

```
outputs.softmax(dim=-1) # softmax sur nbclass dans batch
```

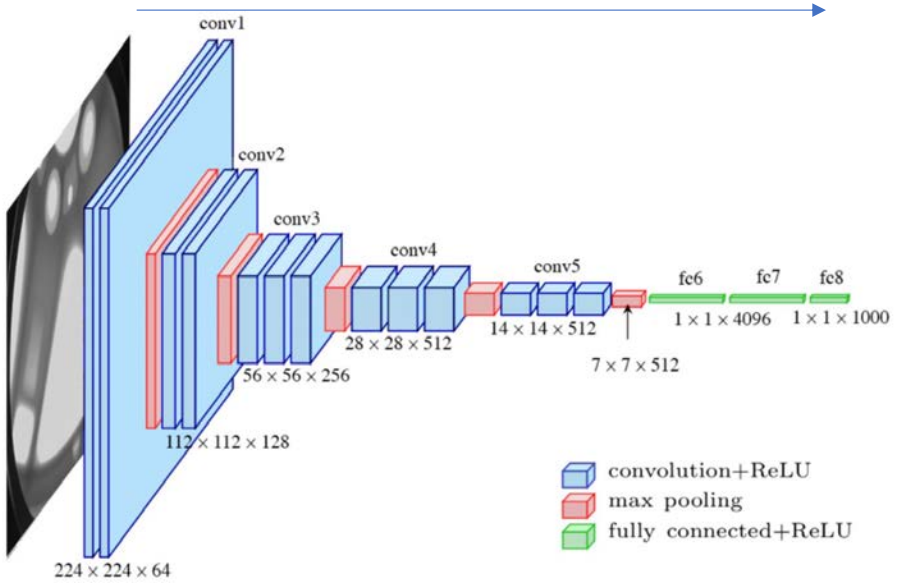
- On peut aussi créer une couche de sotfmax dans un réseau :

```
self.softmax = nn.Softmax(dim=1)
```

https://medium.com/@danushidk507/max-poling-ef545993b6e4

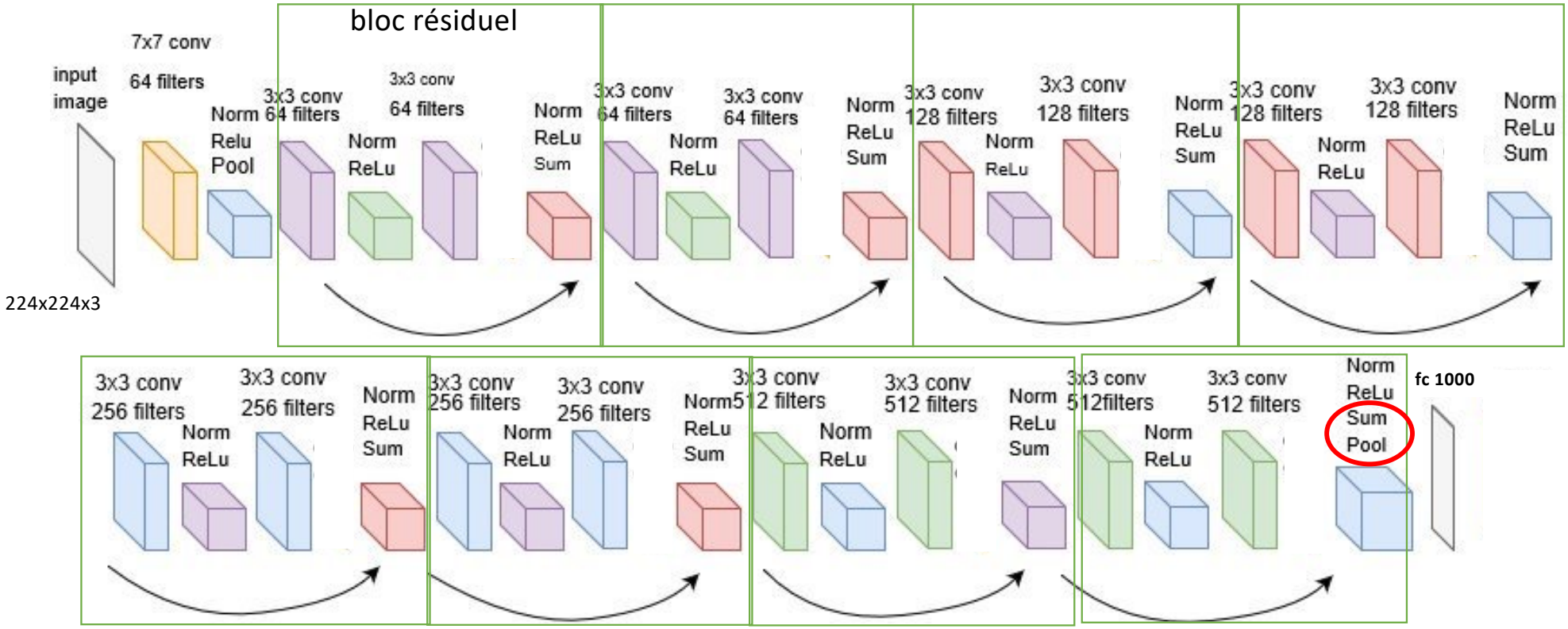
Architectures classiques de réseaux convolutionnels

- Au début (côté images d'entrées) des couches de détections de caractéristiques visuelles sur l'image
  - des convolutions 2D / RELU / MaxPooling/etc.
- A la fin (côté classification) une/plusieurs couches linéaires (*fully connected*) et possible un Softmax pour l'adaptation à la tâche
  - Linear/RELU
- Ex. VGG16 sur ImageNet1K
  - Images 224x224x3
  - 16 couches avec des poids



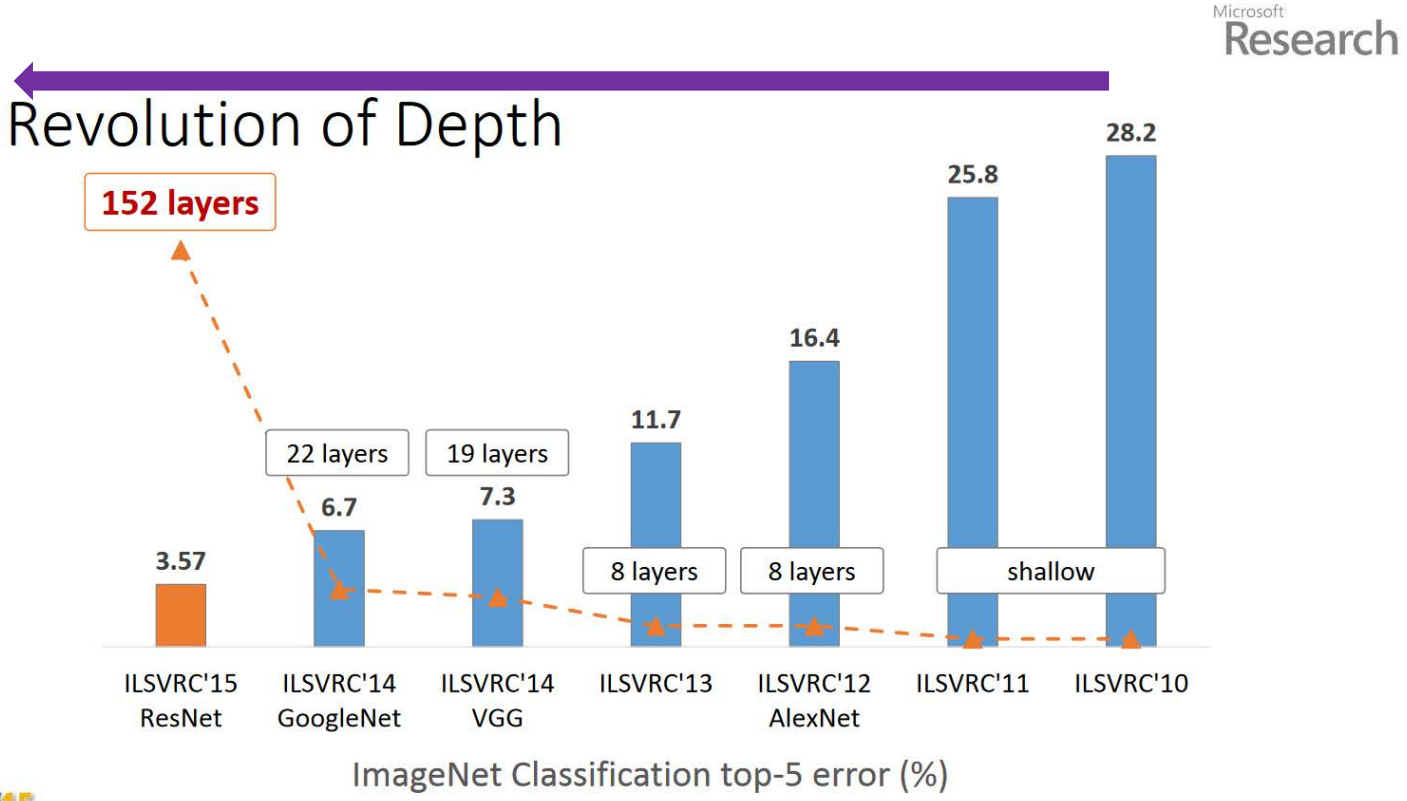
Architectures classiques de réseaux convolutionnels

- ResNet18 (Residual Net, 18 couches) utilisé en TP MM



https://www.researchgate.net/figure/Proposed-Modified-ResNet-18-architecture-for-Bangla-HCR-In-the-diagram-conv-stands-for_fig1_323063171

Les réseaux utilisés sont de plus en plus profonds pour les images



ICCV15 International Conference on Computer Vision

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015.

Apprentissage supervisé – principes 1/2

- Technique d'apprentissage automatique
  - créer une fonction cible à partir de données d'apprentissage. (fonction = paramètres du réseau)
- Données d'apprentissage
  - paires d'objets d'entrée (généralement des vecteurs) et de sorties souhaitées.
- Sortie de la fonction
  - peut être une valeur (régression) continue ou plusieurs étiquettes de classe (classification) de l'objet d'entrée.

- Tâche de l'apprenant supervisé (réseau)
  - prédire la valeur de la fonction pour tout objet d'entrée valide, après avoir vu un certain nombre d'exemples d'apprentissage.

Apprentissage supervisé – principes 2/2

- L'apprenant doit généraliser à partir des données présentées à des situations inédites de manière « raisonnable »

- La tâche parallèle en psychologie humaine et animale est souvent appelée apprentissage conceptuel (dans le cas de la classification)

- Le plus souvent, l'apprentissage supervisé génère un modèle global (end to end) qui permet de faire correspondre les objets d'entrée aux sorties souhaitées

Apprendre une “target function” : formalisation

- Fonction cible : $f : X \rightarrow Y$
  $x \rightarrow y = f(x)$
  - x : entrée, e.g., image en couleur
  - y : sortie désirée, e.g., label
  - X : ensemble d’objets valides en entrée
  - Y : ensemble des valeurs possibles en sortie

f  = “cat”

f  = “dog”

f  = “car”

X : ensemble des images en couleur possibles

$$X = \bigcup_{(w,h)\in\mathbb{N}^{*2}} [0,1]^{w\times h\times 3}$$

Y : ensemble des labels

$$Y = \{\text{“cat”}, \text{“dog”} \ \ldots\}$$

Apprendre une target function : principe

- Function cible : $f : X \rightarrow Y$

$$x \rightarrow y = f(x)$$

  - x : entrée, e.g., image en couleur
  - y : sortie désirée, e.g., label
  - X : ensemble d'objets valides en entrée
  - Y : ensemble des valeurs possibles en sortie (vecteurs)

f  = 0.90 ← "cat", 0.04 ← "dog", 0.01 ← "car", ... ← ...

f  = 0.07, 0.88, 0.02, ...

f  = 0.02, 0.03, 0.86, ...

X : ensemble des images en couleur possibles

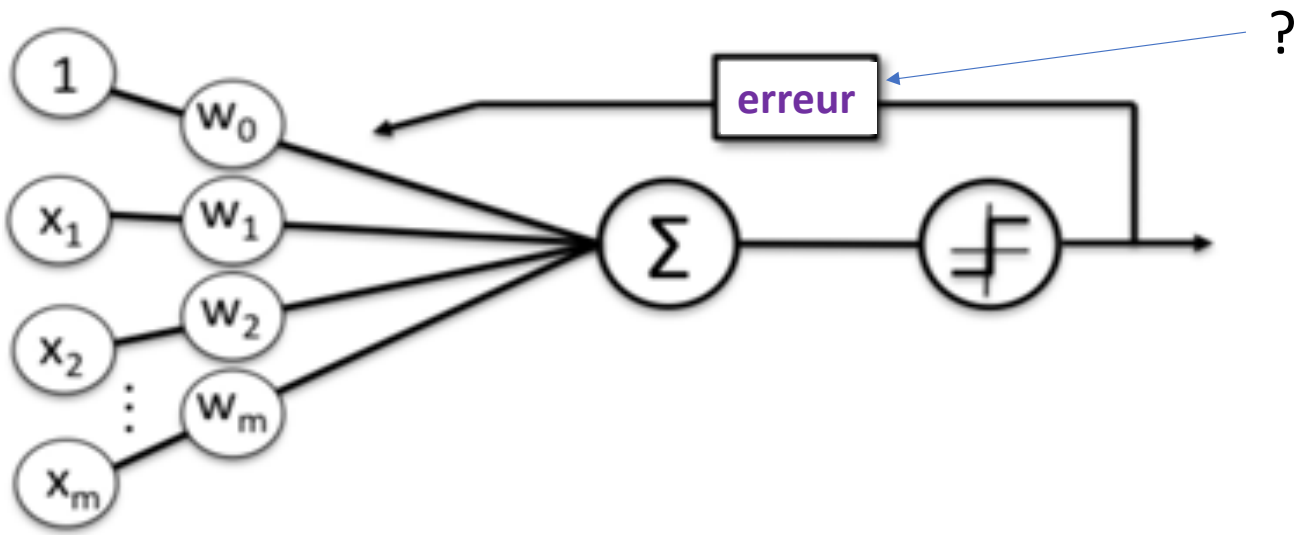
$$X = \bigcup_{(w,h)\in\mathbb{N}^{*2}} [0,1]^{w\times h\times 3}$$

Y : ensemble des labels (vecteur de taille *c*):

$$Y = \mathbb{R}^{|\{\text{"cat","dog" ...}\}|} = \mathbb{R}^{c}$$

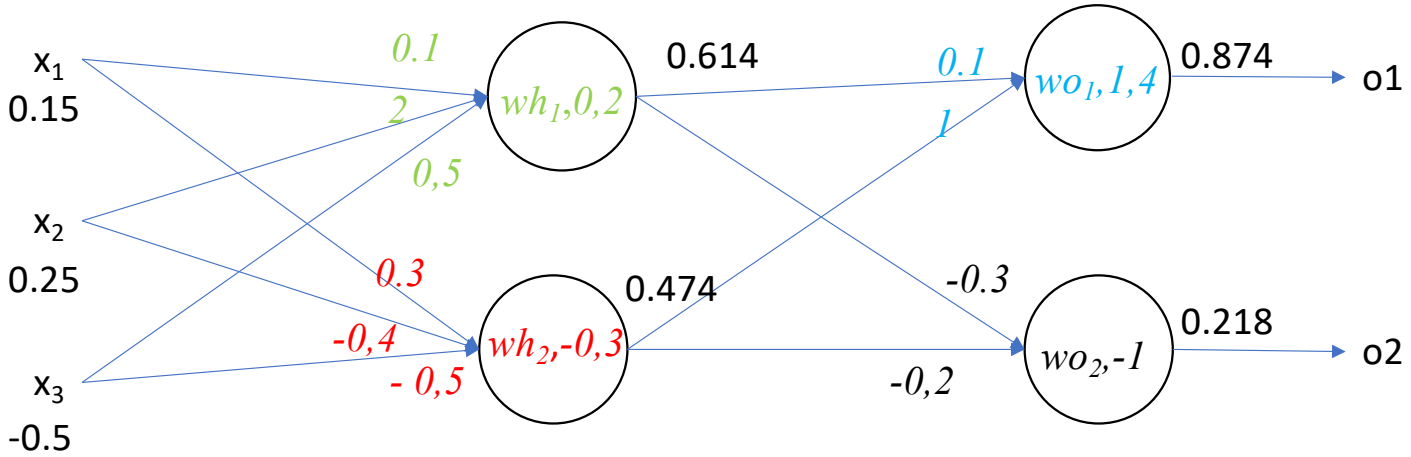
Apprendre une target function : loss / rétro-propagation

- Théorie :
  - On entre une image dans le réseau, et on regarde la sortie générée (inférence)
  - On calcule l'erreur (loss) entre la sortie générée et la « vraie » sortie attendue
  - L'erreur rétro-propagée (backpropagation) sur les paramètres du réseau
    - Utilisation des dérivées partielles (**gradients**) des fonctions des couches (sommes pondérées, activations, convolutions) par rapport à tous les paramètres (poids, biais) en fonction de l'erreur



Apprendre une target function : exemple

Sur les exemples d'apprentissage, on calcule l'erreur entre la prédiction et les valeurs vraies, puis on adapte les poids du réseau en fonction de cette erreur



Reprenons notre réseau avec les calculs précédents : out_{o1} = 0.874 , et out_{o2} = 0.218, noté P=(0.874, 0.218)

- Supposons que dans notre réseau la sortie o1 correspond à chien (classe 1), et o2 à chat (classe 2)
- Supposons que dans notre exemple d'apprentissage en entrée (0.15, 0.25, -0.5) correspond à un chien, c'est-à-dire un vecteur y=(1, 0)

=> On va calculer l'erreur entre la réponse P et la vérité attendue (1,0)

Apprendre une target function : ex. de loss

Erreur (loss) calculée dans notre cas

(ATTENTION : Une formule est adaptée à une tâche)

Pour une tâche de classification multi-classes, la loss classique est l'entropie croisée :

(Cross Entropy)

$$CELoss = -\sum_{c=1}^{N} y_c \log P_c$$

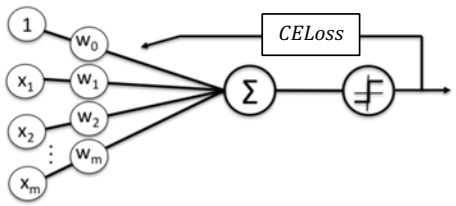
avec : $\log$ = *log népérien*

P_c la prédiction du réseau

y_c la vérité terrain

Avec $P_1 = 0.874$, et $P_2 = 0.218$, et la vérité-terrain $y_1 = 1$, et $y_2 = 0$

$$CELoss = -(1 * \log(0.874) + 0 * \log(0.218))$$
$$= \mathbf{0.05858}$$



Dans pyTorch la cross entropy est définie par : `criterion = nn.CrossEntropyLoss()`

Cette erreur est utilisée par l'*optimizer* pour mettre à jour les poids du réseau.

Apprendre une target function : optim.SGD

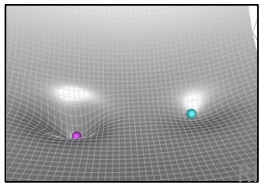
- L'optimiseur (processus d'apprentissage) : modifie les poids du réseau en se basant sur la loss
  - Dérivées partielles, chain rule, dérivées de compositions utilisées pour calculer le gradient (direction, force) de la correction à effectuer (on ne s'intéresse pas aux maths ici, voir fin du cours)

- Exemple d'optimiseur : Stochastic Gradient Descent (SGD)
  - Met à jour les poids θ du réseau avec la formule : $\theta = \theta - \eta * \Delta$
    - Le gradient Δ : dérivée partielle de l'erreur par rapport aux poids du réseau
    - Utilise $' - '$ car le gradient indique l'augmentation de l'erreur (dérivée positive), et on veut la diminuer
    - Le *learning rate*, η, indique la « force » de la mise à jour des poids du réseau
  - *Stochastic* parce qu'on met à jour sur un *batch* du train et pas tout l'ensemble train
    - Bien adapté à une implantation rapide sur GPU
    - 1 batch est un sous-ensemble des données (apprentissage ou test)

- SGD dans pyTorch :

```
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

  - `lr` : learning rate (η dans la formule)
  - `momentum` : inertie pour la mise à jour des paramètres (pour pouvoir sortir d'un minimum local)



https://paperswithcode.com/method/sgd-with-momentum

Apprendre une target function

- Quand on décrit la formule de mise à jour des poids dans le SGD par :

$$\theta = \theta - \eta * \Delta$$

- θ est une notation qui englobe pour TOUS les paramètres à apprendre (cf. fin du cours pour les calculs sur un exemple à… 14 paramètres…)
  - ResNet18 : 11M (Millions de paramètres (nombres réels))
  - ResNet50 : 25M
  - ResNet101 : 44M
  - ResNet152 : 60M

- Ceci explique pourquoi l'apprentissage de ces modèles est très long à faire (énergivore), car pour stabiliser l'apprentissage il faut beaucoup d'exemples…

Apprendre une target function : les époques

Pour qu'un réseau apprenne bien, il lui faut beaucoup d'exemples

- Avec un ensemble de *train* (même gros)
  - On passe plusieurs fois toutes les images de cet ensemble
- Un passage de tout l'ensemble de train = une époque (*epoch*)

C'est ce que qu'on fait dans le TPs avec la fonction

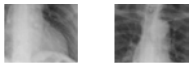
`train_model(…)`

qui utilise la `loss` et l'`optimizer`

Apprendre une target function – Diversité du train

- Agrandir la diversité de l'ensemble d'apprentissage en modifiant de manière réaliste les images du *train*

=> On effectue des transformations aléatoires

  - Rotations : degrés de rotation 
  - Crop : à une position random 
  - Contraste/luminosité/saturation/hue : variation random dans une plage 
  - Flip horizontal/vertical : probabilité 

- L'objectif est d'avoir une généralisation meilleure du réseau par une variabilité accrue des données en empilant les transformations

- Exemple pytorch sur des rotations (50% de chance de flip) :

```
transforms.RandomVerticalFlip(p=0.5)
```

Apprendre une target function – Batch normalization

- Modifier les valeurs en sortie d'une couche
  - Normalise les sorties pour éviter la propagation de très grandes valeurs, sur un batch (cf. intro)
    - » Normalisation sur batch $z_{norm} = \frac{z-\mu}{\sigma}$ (μ = Moyenne, σ= écart-type)
    - » $z_{final} = \gamma . z_{norm} + \beta$, avec γ, β params appris (scale + shift)
      - Durant le train (`model.train()`) : μ et σ calculés par batch
      - Durant l'inférence (`model.test()`) : μ et σ moyens utilisés

  ⇒ Utilisation comme une couche du réseau (nb entrées = nb sorties) :

  ⇒ après une Conv2d (`BatchNorm2d`) ou une couche linéaire (`BatchNorm1d`)

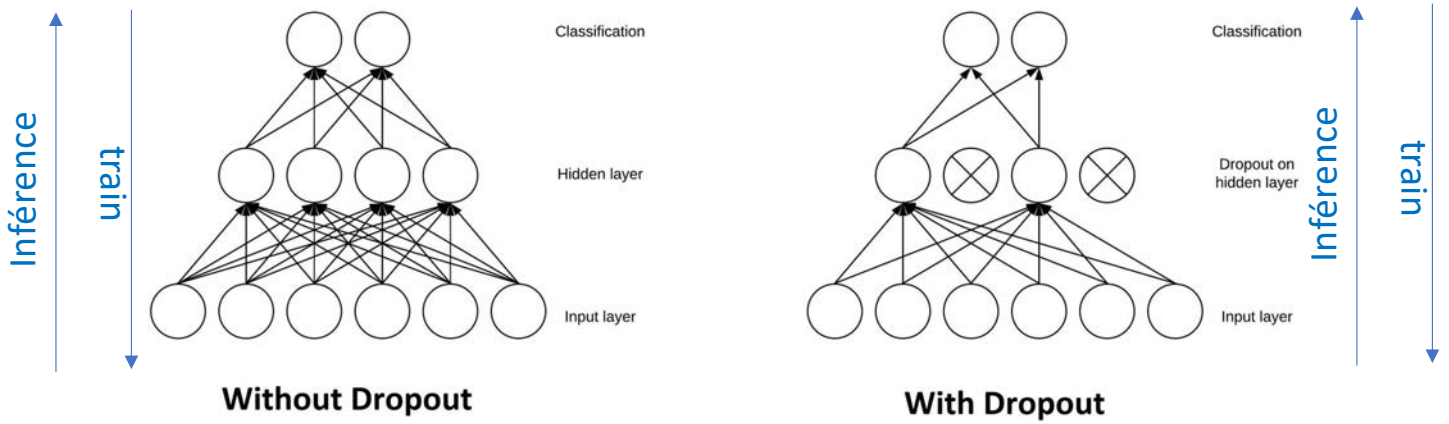
  ⇒ avant la fonction d'activation

  ⇒ Améliore l'apprentissage (apprend plus vite) en contrôlant mieux les échelles de valeurs

    - » Ex. à la sortie d'une conv2D avec 16 canaux en avant une RELU : `nn.BatchNorm2d(16)`

Apprendre une target function – Dropout 1/2

- Dropout : geler des poids dans certaines couches à l'apprentissage
  - Classiquement l'inférence et l'apprentissage utilisent tous les neurones



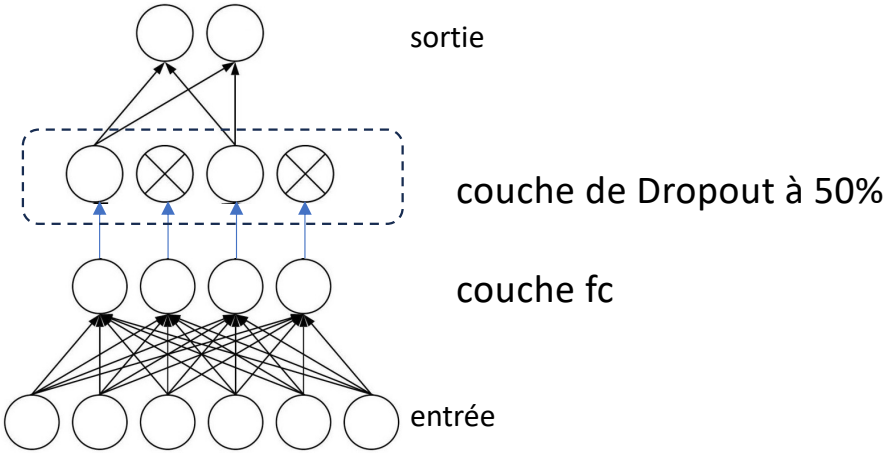
  - Le Dropout, durant le train, oublie certains neurones ⊗ :
    - » A l'inférence du train les noeuds figés ne sont pas utilisés (sortie = 0)
    - » Ces noeuds figés ne rétropropagent pas l'erreur
  - Améliore la généralisation, et d'éviter des co-apprentissages

https://wenkangwei.github.io/2020/11/13/DL-DropOut/

Apprendre une target function – Dropout 2/2

- Hyper-paramètre : proportion de noeuds impactés (~25% (convolutions) et 50% (fc))

- Intégré dans Pytorch par une couche de type *Dropout*



- Ex. Dropout avec probabilité de gel de 20% des noeuds `nn.Dropout(0.2)`
- On active le Dropout durant le train avec `model.train()`
- On désactive le Dropout dans le test avec `model.eval()`

Expérimenter un réseau pour une tâche

- Pour une tâche sur laquelle on a un ensemble de train+validation
  1. On définit un réseau avec une architecture initiale (ex. TP)
  2. On l'apprend sur l'ensemble de *train* (avec quelques dizaines/centaines d'époques) et on teste (inférence) sur l'ensemble de validation
  3. On fait des variations en modifiant des éléments divers
     1. Réseau (ajout/retrait de couches/dropout/…)
     2. Optimizer / modif critère / geler des couches
     3. Taille des batches
     4. Pré-traitements
  4. On compare la qualité d'inférence par rapport au réseau initial sur la validation et on regarde ce qui marche mieux… *du feeling dirigé*

  => Cela peut être très gourmand, en temps/électricité

Passer la sortie d'une couche de convolution à une couche fully connected (NEW)

- Retour sur notre réseau des TPs, entre couche 5_2 et fc :

```
…
self.layer5_2 = nn.Sequential(
nn.MaxPool2d(kernel_size=2, stride=2))  # on sort un volume : 4 x 4 x 64

self.fc = nn.Sequential(
nn.Linear(64 * 4 * 4, 128),
nn.ReLU(),
nn.Linear(128, 128),
nn.ReLU(),
nn.Linear(128, num_classes)
)
…
x = self.layer5_2(x)
x = x.view(x.size(0), -1)
x = self.fc(x)
…
```

`x.size(0)` : taille du batch (car traitement en parellèle sur toutes les images du batch)

`-1` : aplatit le volume : 4*4*64 en un vecteur 1024D

Réutiliser un réseau existant pour une autre tâche

- Supposons qu'un réseau convolutionnel R avec une architecture classique (convolutions, puis fc) fonctionne très bien pour une tâche de classification des oiseaux (100 classes)
- Peut-on le « réutiliser » pour l'appliquer à un autre cas, par exemple classifier, parmi 50 classes, des images de chiens ?

  => OUI

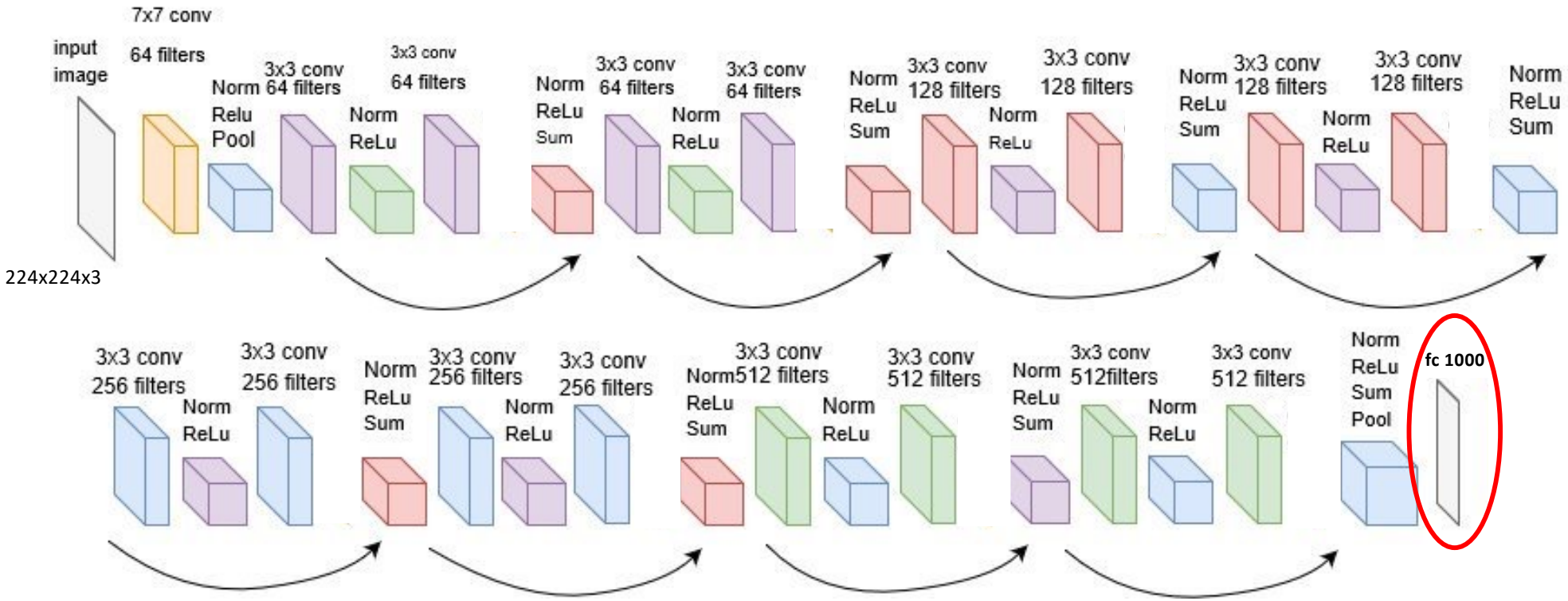
  – C'est ce que fait le transfert learning

Transfert learning

- Principe
  - Réutiliser un réseau déjà appris sur d'autres datasets (appelé backbone)
  - Ce réseau est déjà optimisé sur des grands datasets (ImageNet par ex.) pour d'autres tâches, on peut en profiter !
- Mise en oeuvre
  - Adaptation d'un réseau en sortie (quasiment obligatoire)
    1. On "coupe la tête" du réseau initial pour profiter des poids appris (sur les parties convolutionnelles) sur de grands datasets
    2. On adapte les dimensions de sortie du backbone
       - Par ex. resnet18 sort un vecteur de 1000D, et on veut 9 classes dans le TP
  - Adaptation des données en entrée (optionnel)
    - » Par ex. VGG16 utilise des images de 224x224x3, et on a des images 28x28x3 en TP

Transfert learning - adaptation

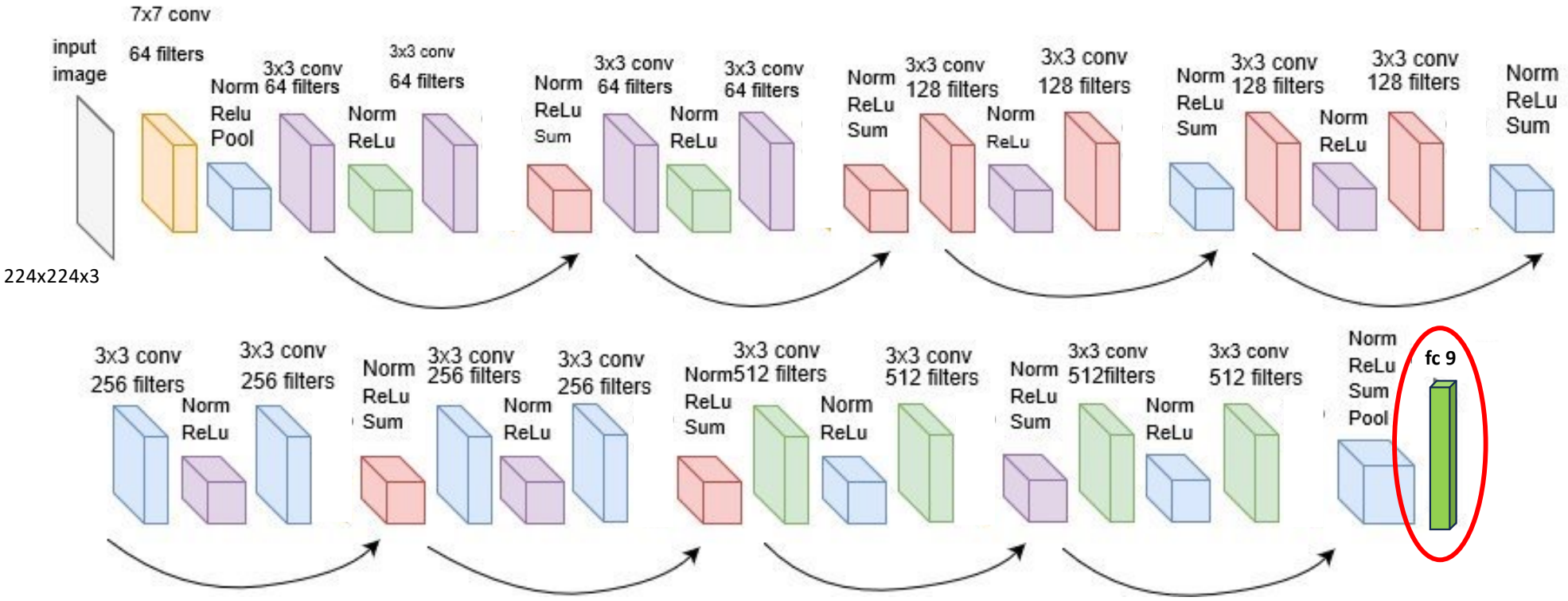
- ResNet18, adapter la sortie



  - ResNet18 fournit en sortie un vecteur de 1000D car il est dédié à ImageNet

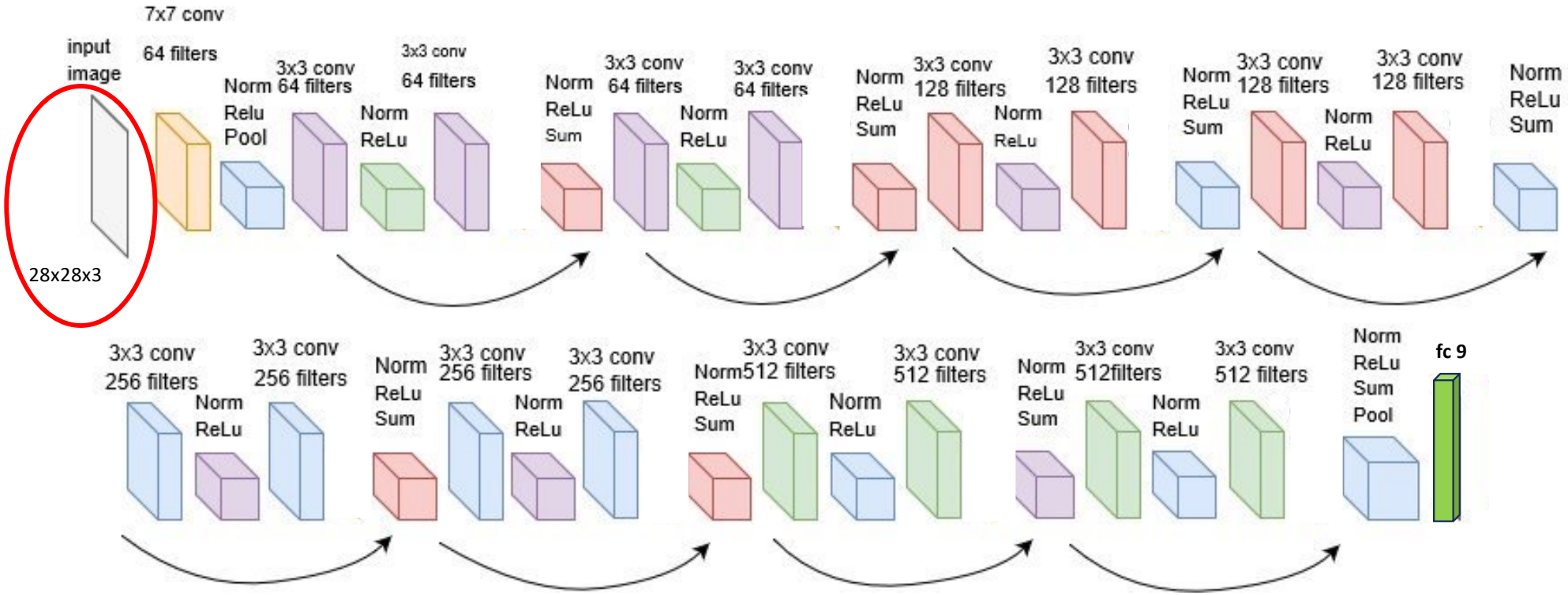
Transfert learning - adaptation

- Adaptation ResNet18 = > ResNet18’
  - ResNet18 fournit en sortie un vecteur de 1000D car il est entraîné sur ImageNet
  - Notre cas => 9 classes, on remplace la couche fc par une de 9 neurones => ResNet18’



Transfert learning - adaptation

- ResNet18, adapter l'entrée ? => avec cette architecture pas nécessaire avec le pooling final



Transfert learning - adaptation

- Création de ResNet18'
  - Importer ResNet18 existant
    - » En pytorch :

```
import torchvision.models as models
…
model_ftres = models.resnet18(pretrained=True) # pour charger les poids existants
# OU model_ftres = models.resnet18(pretrained=False) # pour ne pas charger les poids
```

  - on peut afficher le réseau

```
print(model_ftres)
```

    - » Cet affichage affiche en particulier

```
(fc): Linear(in_features=512, out_features=1000, bias=True)
```

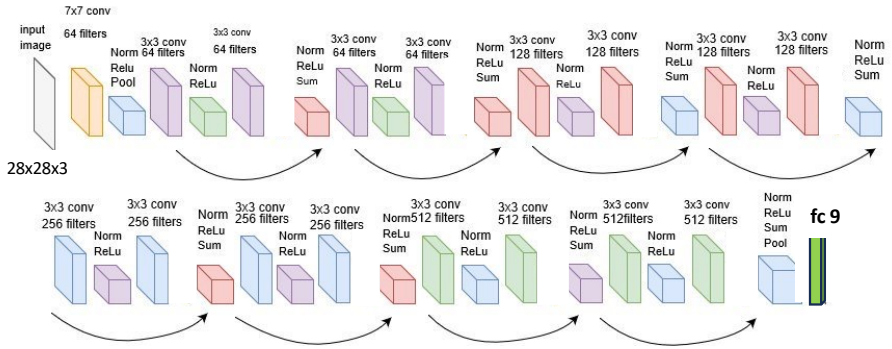
  - Modifier le réseau, on va réaffecter la couche **fc**

```
model_ftres.fc = nn.Sequential(
        nn.Linear(512, 9)) # on coupe la tête et on met un fully connected avec 9 sorties
```

  => Le réseau est maintenant modifié pour avoir 9 neurones en sortie.

Transfert learning - adaptation

- Apprentissage de ResNet18'
  1. Modification du réseau (tranparents précédents)
  2. Comment apprendre
     1. On ré-apprend, from scratch, tous les poids (convolutions de ResNet18 + la couches fc) : beaucoup de paramètres…
     2. Fine-tuning v1 : On adapte tous les poids (convolutions de ResNet18 + la couches fc)
     3. Fine-tuning v2 : on fige les poids des convolutions de ResNet18 et on apprend les poids de nos couches fc
- On va tester les 2 fine-tuning en TP



Transfert learning – fine-tuning v1

- Apprentissage de ResNet18' avec modification de tous les poids pré-appris de resnet18
  - Apprentissage
    - » Par défaut, on redéclare un loss et et un optimizer (comme précédemment)

```
criterionRes = nn.CrossEntropyLoss()
optimizerRes = optim.SGD(model_ftres.parameters(), lr=lr,
                         momentum=0.9)
```

  - On relance l'apprentissage comme précédemment

```
train_model(model=model_ftres, dataloader=train_loader,
                  criterion=criterionRes, optimizer=optimizerRes,
                  num_epochs=1, device=device)
```

    - » Note : **num_epoch = 1** car l'apprentissage est long +/- 20 minutes (cf. TP)
  - On refait le test comme précédemment, avec le bon réseau **model_ftres**

Transfert learning – fine-tuning v2

- Si on a chargé ResNet18, dans **`model_ftres2`** et modifié la couche **`fc`**
- Pour geler certaines couches d'un réseau pour ne pas modifier leurs poids durant l'apprentissage
  1. On gèle toutes les couches

```
for param in model_ftres2.parameters():
        param.requires_grad = False
```

  2. On dégèle les paramètres de la couche voulue, par ex. **`fc`**

```
for param in model_ftres2.fc.parameters():
        param.weight.requires_grad = True
```

- On redéclare le loss et l'optimizer, puis on relance le train et le test comme pour l'adaptation 1.

Lien avec l'indexation et la recherche multimédia

- Si on veut classifier un ensemble d'images E à analyser pour faire de la recherche d'information (par ex. nos photos personnelles, avec 10 classes)
  1. On fait apprendre et/ou on fine-*tune* un réseau pour l'apprendre sur les classes possibles sur un ensemble d'apprentissage
  2. On applique le réseau en inférence sur les images de test de E on stocke la/les meilleurs labels par image (avec éventuellement leur poids)
  3. On peut rechercher ensuite les images en faisant des requêtes sur les labels stockés, comme avec un modèle vectoriel sur le texte

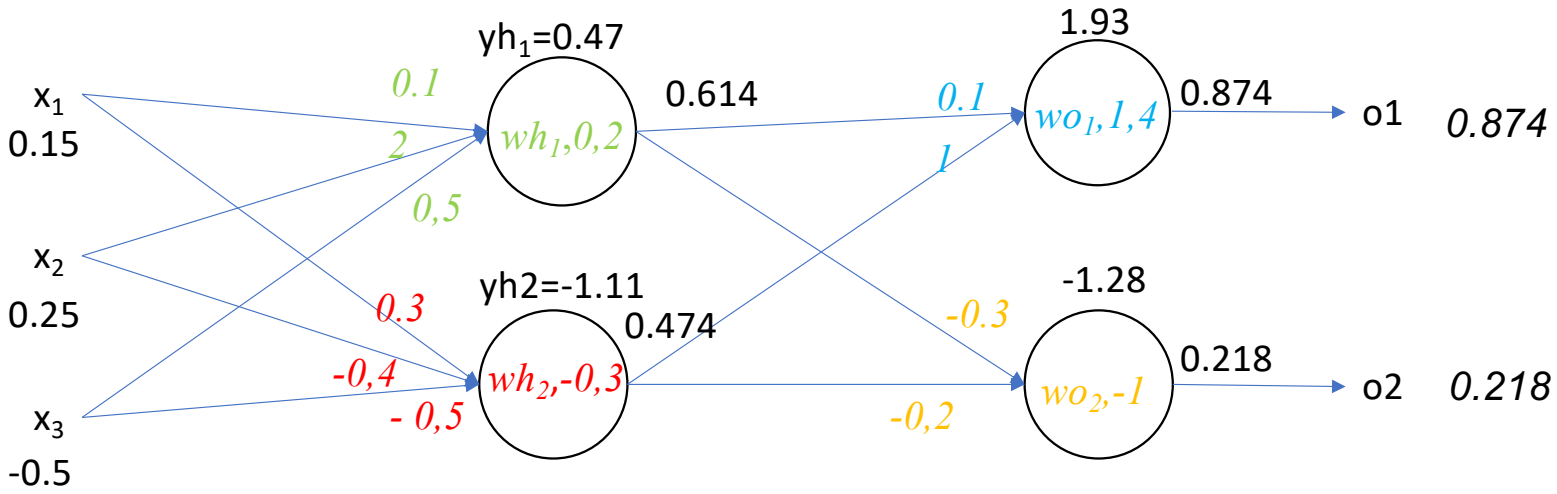
Conclusion

- Ce qu'on sait
  - Comment est fait un réseau convolutionnel (couches)
  - Quelles sont les couches les plus courantes (convolutions, …)
  - Comment on apprend à partir d'exemples pour la classification automatique
  - Comment « *tuner* » un réseau
  - Comment on peut réutiliser un réseau
  - Comment travailler sur Pytorch (TPs)

- Ce que l'on ne sait pas
  - Les détails de l'apprentissage (gradients, backpropagation (cf. la suite))
    - Ce n'est pas obligatoire, à part pour ceux qui développent les librairies de Deep Learning
  - Comment définir un data loader dans pytorch
    - ce n'est pas grave pour créer/utiliser des réseaux sur des corpus existants pour lesquels un tel loader existe (cf. TPs)
    - Mais on sait modifier les images en entrée pour diversifier le dataset

Backpropagation : exemple 1/24

Sur un exemple d'apprentissage, on calcule l'erreur entre la prédiction et les valeurs vraies, puis on adapte les poids du réseau en fonction de cette erreur



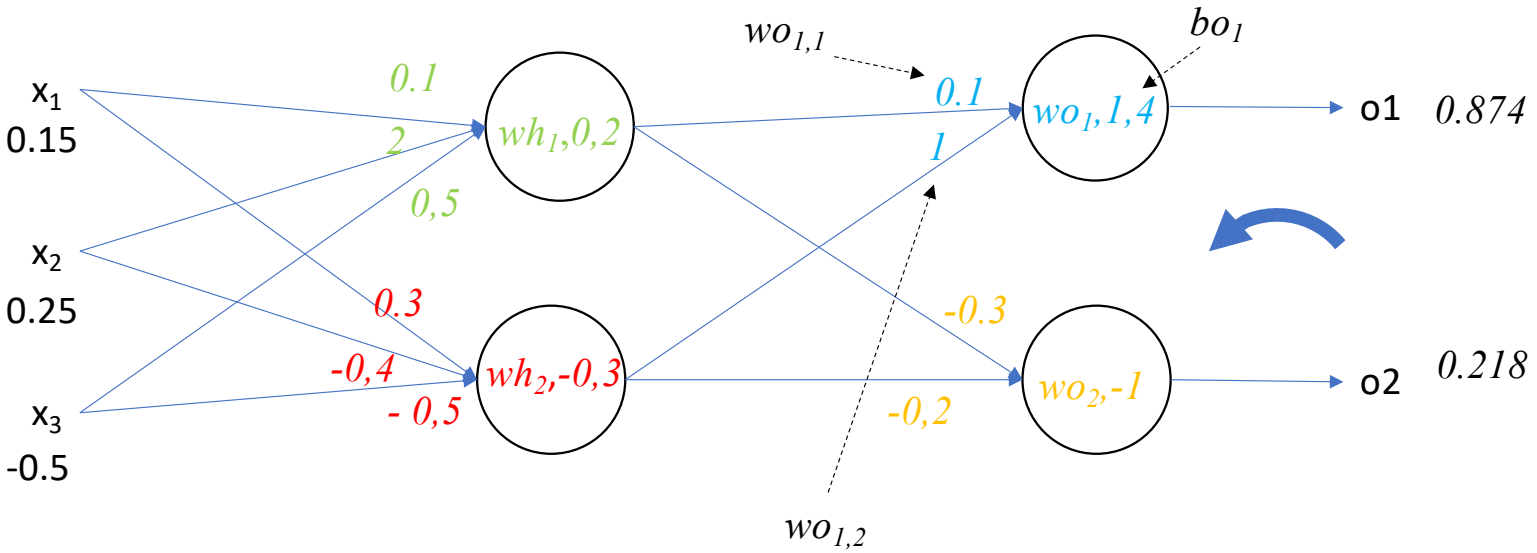
Reprenons notre réseau avec les calculs précédents : o1 = 0.874 , et o2 = 0.218, noté P=(0.874, 0.218)

- Supposons que la sortie o1 correspond à « chien » (classe 1), et o2 à « chat » (classe 2)
- Supposons que dans l'exemple d'apprentissage en entrée est <(0.15, 0.25, -0.5), chien>, c'est-à-dire par rapport à notre réseau un vecteur y=(1, 0)

=> L'erreur entre la réponse P et la vérité attendue (1,0) est : PCELoss $E_{tot} = E_{o1} + E_{o2} = 0.05858$

Backpropagation : exemple 2/24

Sur l'entrée d'apprentissage, on adapte les poids du réseau en fonction de l'erreur calculée par rétropropagation du gradient en corrigeant chaque poid du réseau par rapport à sa contribution. Cette contribution est calculée par la dérivée partielle de l'erreur par rapport au poids considéré.



CELoss $E_{tot} = E_{o1} + E_{o2} = 0.05858$

Dérivées partielles pour de l'erreur par rapport à $wo_{1,1}$:

$$\frac{\partial E_{tot}}{\partial wo_{1,1}} = \frac{\partial E_{tot}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial y_{o1}} * \frac{\partial y_{o1}}{\partial wo_{1,1}}$$

Backpropagation : exemple 3/24

Dérivées partielles pour $wo_{l,l}$:

$$\frac{\partial E_{tot}}{\partial wo_{1,1}} = \frac{\partial E_{tot}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial y_{o1}} * \frac{\partial y_{o1}}{\partial wo_{1,1}}$$

$$\frac{\partial E_{tot}}{\partial out_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} + \frac{\partial E_{o2}}{\partial out_{o1}} = \frac{\partial}{\partial out_{o1}} - (v1.\log(out_{o1})) + \frac{\partial}{\partial out_{o1}} - (v2.\log(out_{o2}))$$

$$\frac{\partial}{\partial out_{o1}} - (v1.\log(out_{o1})) = -\frac{v1}{out_{o1}} = -1.14 \quad \text{et} \frac{\partial}{\partial out_{o1}} - (v2.\log(out_{o2})) = 0 \text{ donc} \frac{\partial E_{tot}}{\partial out_{o1}} = -1.14$$

$$\frac{\partial out_{o1}}{\partial y_{o1}} = \frac{\partial \sigma(y_{o1})}{\partial y_{o1}} = \frac{\partial}{\partial y_{o1}} \frac{1}{1+e^{-y_{o1}}} = \frac{e^{-y_{o1}}}{(1+e^{-y_{o1}})^2} = \frac{e^{-1.93}}{(1+e^{-1.93})^2} = 0.1103$$

$$\frac{\partial y_{o1}}{\partial wo_{1,1}} = \frac{\partial}{\partial wo_{1,1}} out_{h1} * wo_{1,1} + out_{h2} * wo_{1,2} + b_{o1} = out_{h1} = 0.614$$

$$\frac{\partial E_{tot}}{\partial wo_{1,1}} = -\frac{v1}{out_{o1}} * \frac{e^{-y_{o1}}}{(1+e^{-y_{o1}})^2} * out_{h1} = -1.14 * 0.1103 * 0.614 = -0.078$$

Backpropagation example 4/24

Dérivées partielles pour $wo_{1,2}$:

$$\frac{\partial E_{tot}}{\partial wo_{1,2}} = \frac{\partial E_{tot}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial y_{o1}} * \frac{\partial y_{o1}}{\partial wo_{1,2}}$$

$\frac{\partial E_{tot}}{\partial out_{o1}} = -1.14$ déjà calculé

$\frac{\partial out_{o1}}{\partial y_{o1}} = 0.1103$ déjà calculé

$$\frac{\partial y_{o1}}{\partial wo_{1,2}} = \frac{\partial}{\partial wo_{1,2}} out_{h1} * wo_{1,1} + out_{h2} * wo_{1,2} + b_{o1} = out_{h2} = 0.438$$

$$\frac{\partial E_{tot}}{\partial wo_{1,2}} = -1.14 * 0.1103 * 0.438 = -0.060$$

Backpropagation : exemple 5/24

Dérivées partielles pour bo_1 :

$$\frac{\partial E_{tot}}{\partial bo_1} = \frac{\partial E_{tot}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial y_{o1}} * \frac{\partial y_{o1}}{\partial bo_1}$$

$\frac{\partial E_{tot}}{\partial out_{o1}} = -1.14$ déjà calculé

$\frac{\partial out_{o1}}{\partial y_{o1}} = 0.1103$ déjà calculé

$$\frac{\partial y_{o1}}{\partial bo_1} = \frac{\partial}{\partial bo_1} out_{h1} * wo_{1,1} + out_{h2} * wo_{1,2} + bo_1 = 1$$

$$\frac{\partial E_{tot}}{\partial bo_1} = -1.14 * 0.1103 * 1 = -0.1262$$

Backpropagation : exemple 6/24

Dérivées partielles pour $wo_{2,1}$:

$$\frac{\partial E_{tot}}{\partial wo_{2,1}} = \frac{\partial E_{tot}}{\partial out_{o2}} * \frac{\partial out_{o2}}{\partial y_{o2}} * \frac{\partial y_{o2}}{\partial wo_{2,1}}$$

$$\frac{\partial E_{tot}}{\partial out_{o2}} = \frac{\partial - (v1.\log(out_{o1}) + v2.\log(out_{o2}))}{\partial out_{o2}}$$

$$\frac{\partial E_{tot}}{\partial out_{o2}} = -\frac{v2}{out_{o2}} = 0$$

Sans calculer les autres dérivées :

$$\frac{\partial E_{tot}}{\partial wo_{2,1}} = 0$$
$$\frac{\partial E_{tot}}{\partial wo_{2,2}} = 0$$
$$\frac{\partial E_{tot}}{\partial bo_{2}} = 0$$

Backpropagation : exemple 7/24

Si on applique SGD (sans momentum, learning rate η=0.001) pour mettre à jours les poids des deux neurones de la couche O :

$$wo_{1,1}{}' = wo_{1,1} - \eta * \frac{\partial E_{tot}}{\partial wo_{1,1}} = 0.1 - 0.001 * (-0.078) = 0{,}100078$$

$$wo_{1,2}{}' = wo_{1,2} - \eta * \frac{\partial E_{tot}}{\partial wo_{1,2}} = 1 - 0.001 * (-0.060) = 1{,}000060$$

$$bo_1{}' = bo_1 - \eta * \frac{\partial E_{tot}}{\partial bo_1} = 1.4 - 0.001 * (-0.1262) = 1{,}400013$$

$$wo'_{2,1} = wo_{2,1} - \eta * \frac{\partial E_{tot}}{\partial wo_{2,1}} = 0.1 - 0.001 * (0) = 0{,}1 = wo_{2,1}$$

$$wo_{2,2}{}' = wo_{2,2}$$

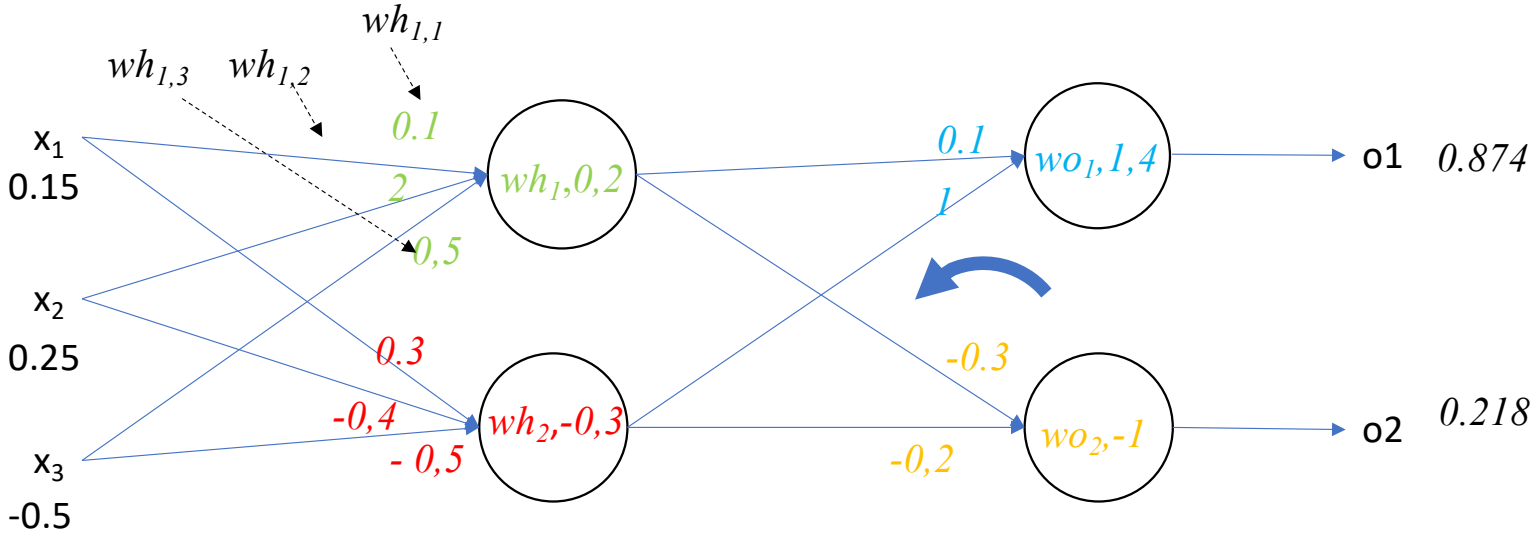
$$bo_2{}' = bo_2$$

Donc, la fonction linéaire (cf. slide 11) mise à jour pour o1 est :

$$yo'_1 = wo'_{1,1} * out_{h1} + wo'_{2,1} * out_{h2} + bo'_1$$

Backpropagation : exemple 8/24

On continue avec la couche H



Dérivées partielles pour h1 :

$$\frac{\partial E_{tot}}{\partial wh_{1,1}} = \frac{\partial E_{tot}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial y_{h1}} * \frac{\partial y_{h1}}{\partial wh_{1,1}}$$

Backpropagation : exemple 9/24

On fait les calculs partie par partie, sachant que la sortie de h1 va vers o1 et o2 :

$$\frac{\partial E_{tot}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial y_{o1}} + \frac{\partial y_{o1}}{\partial out_{h1}}$$

Or on a déjà calculé : $\frac{\partial E_{o1}}{\partial y_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial y_{o1}} = -1.14 * 0.1103 = -0.1262$

Pour : $\frac{\partial y_{o1}}{\partial out_{h1}} = \frac{\partial}{\partial out_{h1}} out_{h1} * wo_{1,1} + out_{h2} * wo_{1,2} + b_{o1}$

Donc $\frac{\partial y_{o1}}{\partial out_{h1}} = wo_{1,1}$=0.1

On obtient finalement : $\frac{\partial E_{o1}}{\partial out_{h1}} = -0.1262 * 0.1 = -0.01262$

Backpropagation : exemple 10/24

$$\frac{\partial E_{tot}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

$$\frac{\partial E_{o2}}{\partial out_{h1}} = \frac{\partial E_{o2}}{\partial y_{o2}} * \frac{\partial y_{o2}}{\partial out_{h1}}$$

On a déjà calculé : $\frac{\partial E_{o2}}{\partial y_{o2}} = \frac{\partial E_{o2}}{\partial out_{o2}} * \frac{\partial out_{o2}}{\partial y_{o2}} = 0$

Donc $\frac{\partial E_{o2}}{\partial out_{h1}} = 0$

Donc $\frac{\partial E_{tot}}{\partial out_{h1}} = -0.01262 + 0 = -0.01262$

Backpropagation : exemple 11/24

On s'intéresse maintenant à : $\frac{\partial out_{h1}}{\partial y_{h1}}$

On a $\frac{\partial out_{h1}}{\partial y_{h1}} = \frac{\partial \sigma(y_{h1})}{\partial y_{h1}} = \frac{\partial}{\partial y_{h1}} \frac{1}{1+e^{-y_{h1}}} = \frac{e^{-y_{h1}}}{(1+e^{-y_{h1}})^2} = \frac{e^{-0.47}}{(1+e^{-0.47})^2} = 0.237$

Il reste uniquement à calculer : $\frac{\partial y_{h1}}{\partial wh_{1,1}} = \frac{\partial}{\partial wh_{1,1}} \mathrm{x}_1 * wh_{1,1} + \mathrm{x}_2 * wh_{1,2} + \mathrm{x}_3 * wh_{1,3} + b_{h1} = \mathrm{x}_1 = 0.15$

On obtient donc : $\frac{\partial E_{tot}}{\partial wh_{1,1}} = \frac{\partial E_{tot}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial y_{h1}} * \frac{\partial y_{h1}}{\partial wh_{1,1}} = -0.01262 * 0.237 * 0.15 = -0.000448$

Backpropagation : exemple 12/24

Pour $wh_{2,1}$:

$$\frac{\partial E_{tot}}{\partial wh_{1,2}} = \frac{\partial E_{tot}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial y_{h1}} * \frac{\partial y_{h1}}{\partial wh_{1,2}}$$

On a déjà calculé $\frac{\partial E_{tot}}{\partial out_{h1}} = -0.01262$, et $\frac{\partial out_{h1}}{\partial y_{h1}} = 0.237$

$$\frac{\partial y_{h1}}{\partial wh_{1,2}} = \frac{\partial}{\partial wh_{1,2}} x_1 * wh_{1,1} + x_2 * wh_{1,2} + x_3 * wh_{1,3} + b_{h1} = x_2 = 0.25$$

Ce qui donne :

$$\frac{\partial E_{tot}}{\partial wh_{1,2}} = \frac{\partial E_{tot}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial y_{h1}} * \frac{\partial y_{h1}}{\partial wh_{1,2}} = -0.01262 * 0.237 * 0.25 = -0{,}000747466$$

Backpropagation : exemple 13/24

Pour $wh_{1,3}$:

$$\frac{\partial E_{tot}}{\partial wh_{1,3}} = \frac{\partial E_{tot}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial y_{h1}} * \frac{\partial y_{h1}}{\partial wh_{1,3}}$$

On a déjà calculé $\frac{\partial E_{tot}}{\partial out_{h1}} = 0.01262$, et $\frac{\partial out_{h1}}{\partial y_{h1}} = 0.237$

$$\frac{\partial y_{h1}}{\partial wh_{1,3}} = \frac{\partial}{\partial wh_{1,3}} x_1 * wh_{1,1} + x_2 * wh_{1,2} + x_3 * wh_{1,3} + b_{h1} = x_3 = -0.5$$

Ce qui donne :

$$\frac{\partial E_{tot}}{\partial wh_{1,3}} = \frac{\partial E_{tot}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial y_{h1}} * \frac{\partial y_{h1}}{\partial wh_{1,3}} = -0.01262 * 0.237 * -0.5 = \textit{0,001494933}$$

Backpropagation : exemple 14/24

Pour bh1,

$$\frac{\partial E_{tot}}{\partial bh_1} = \frac{\partial E_{tot}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial y_{h1}} * \frac{\partial y_{h1}}{\partial bh_1}$$

On a déjà calculé $\frac{\partial E_{tot}}{\partial out_{h1}} = -0.01262$, et $\frac{\partial out_{h1}}{\partial y_{h1}} = 0.237$

$$\frac{\partial y_{h1}}{\partial bh_1} = \frac{\partial}{\partial bh_1} x_1 * wh_{1,1} + x_2 * wh_{1,2} + x_3 * wh_{1,3} + b_{h1} = 1$$

Ce qui donne :

$$\frac{\partial E_{tot}}{\partial bh_1} = \frac{\partial E_{tot}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial y_{h1}} * \frac{\partial y_{h1}}{\partial bh_1} = -0.01262 * 0.237 * 1 = -0{,}002989866$$

Backpropagation : exemple 15/24

Si on utilise SGD pour modifier les poids de h1 :

$$wh'_{1,1} = wh_{1,1} - \eta * \frac{\partial E_{tot}}{\partial wh_{1,1}} = 0.1 - 0.001*(-0.000448) = 0.000000448$$

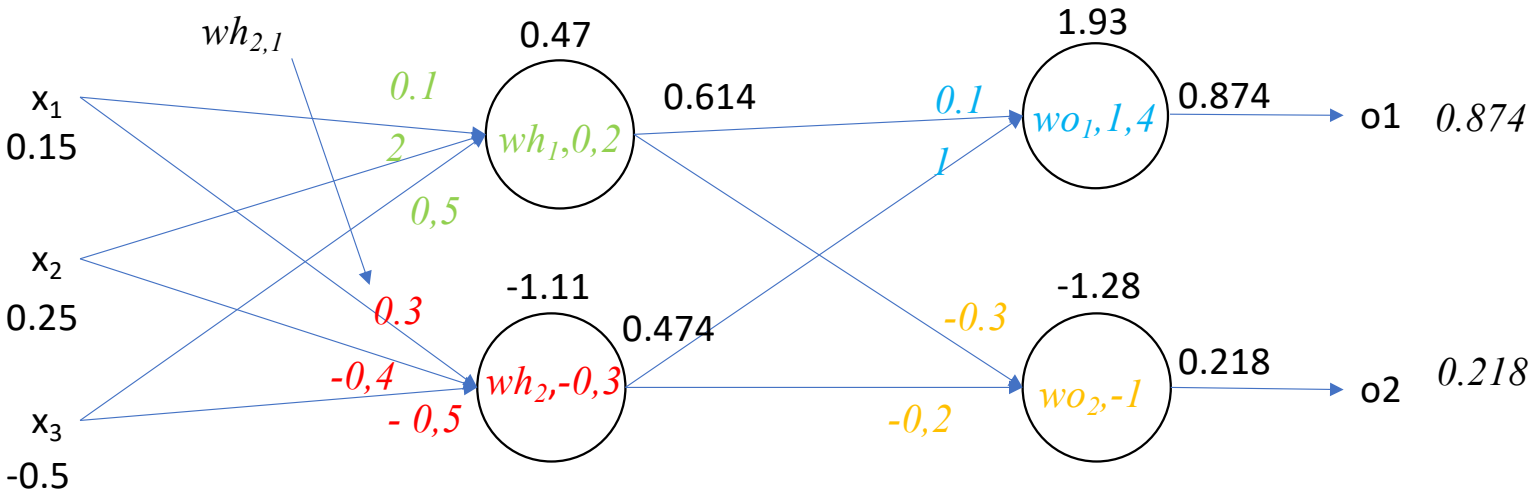
$$wh'_{1,2} = wh_{1,2} - \eta * \frac{\partial E_{tot}}{\partial wh_{1,2}} = 2 - 0.001 * (-0.000747466) = 2.000000747$$

$$wh'_{1,3} = wh_{1,3} - \eta * \frac{\partial E_{tot}}{\partial wh_{1,3}} = 0.5 - 0.001 * (0{,}001494933) = 0{,}499998505$$

$$bh'_{1} = bh_{1} - \eta * \frac{\partial E_{tot}}{\partial bh_{1}} = 0.2 - 0.001 * (-0{,}002989866) = 0{,}20000299$$

Backpropagation : exemple 16/24

On continue avec le neurone h2 de la couche H



Dérivées partielles pour h2,1 :

$$\frac{\partial E_{tot}}{\partial wh_{2,1}} = \frac{\partial E_{tot}}{\partial out_{h2}} * \frac{\partial out_{h2}}{\partial y_{h2}} * \frac{\partial y_{h2}}{\partial wh_{2,1}}$$

Backpropagation : exemple 17/24

On fait les calculs partie par partie, sachant que la sortie de h1 va vers o1 et o2 :

$$\frac{\partial E_{tot}}{\partial out_{h2}} = \frac{\partial E_{o1}}{\partial out_{h2}} + \frac{\partial E_{o2}}{\partial out_{h2}}$$

$$\frac{\partial E_{o1}}{\partial out_{h2}} = \frac{\partial E_{o1}}{\partial y_{o1}} + \frac{\partial y_{o1}}{\partial out_{h2}}$$

Or on a déjà calculé : $\frac{\partial E_{o1}}{\partial y_{o1}} = -0.1262$

Pour : $\frac{\partial y_{o1}}{\partial out_{h2}} = \frac{\partial}{\partial out_{h2}} out_{h1} * wo_{1,1} + out_{h2} * wo_{1,2} + b_{o1} = wo_{1,2} = 1$

On obtient finalement : $\frac{\partial E_{o1}}{\partial out_{h2}} = -0.1262 * 1 = -0.1262$

Pour $\frac{\partial E_{o2}}{\partial out_{h2}} = \frac{\partial E_{o2}}{\partial y_{o2}} * \frac{\partial y_{o2}}{\partial out_{h2}}$ on a déjà calculé : $\frac{\partial E_{o2}}{\partial y_{o2}} = \frac{\partial E_{o2}}{\partial out_{o2}} * \frac{\partial out_{o2}}{\partial y_{o2}} = 0$

Donc $\frac{\partial E_{o2}}{\partial out_{h2}} = 0$

Ce qui donne $\frac{\partial E_{tot}}{\partial out_{h2}} = -0.1262 + 0 = -0.1262$

Backpropagation : exemple 18/24

On calcule :

$$\frac{\partial out_{h2}}{\partial y_{h2}} = \frac{\partial \sigma(y_{h2})}{\partial y_{h2}} = \frac{\partial}{\partial y_{h2}} \frac{1}{1+e^{-y_{h2}}} = \frac{e^{-y_{h2}}}{(1+e^{-y_{h2}})^2} = \frac{e^{+0.11}}{(1+e^{+0.11})^2} = 0.25$$

Il reste uniquement à calculer :

$$\frac{\partial y_{h2}}{\partial wh_{2,1}} = \frac{\partial}{\partial wh_{2,1}} \mathrm{x}_1 * wh_{2,1} + \mathrm{x}_2 * wh_{2,2} + \mathrm{x}_3 * wh_{2,3} + b_{h2} = \mathrm{x}_1 = 0.15$$

On obtient donc :

$$\frac{\partial E_{tot}}{\partial wh_{2,1}} = \frac{\partial E_{tot}}{\partial out_{h2}} * \frac{\partial out_{h2}}{\partial y_{h2}} * \frac{\partial y_{h2}}{\partial wh_{2,1}} = -0.1262 * 0.25 * 0.15 = -0.004719$$

Backpropagation : exemple 19/24

Dérivées partielles pour $wh_{2,2}$:

$$\frac{\partial E_{tot}}{\partial wh_{2,2}} = \frac{\partial E_{tot}}{\partial out_{h2}} * \frac{\partial out_{h2}}{\partial y_{h2}} * \frac{\partial y_{h2}}{\partial wh_{2,2}}$$

On a déjà calculé : $\frac{\partial E_{tot}}{\partial out_{h2}} = -0.1262$ et $\frac{\partial out_{h2}}{\partial y_{h2}} = 0.25$

Il reste à calculer : $\frac{\partial y_{h2}}{\partial wh_{2,2}} = \frac{\partial}{\partial wh_{2,2}} x_1 * wh_{2,1} + x_2 * wh_{2,2} + x_3 * wh_{2,3} + b_{h2} = x_2 = 0.25$

On obtient donc :

$$\frac{\partial E_{tot}}{\partial wh_{2,2}} = \frac{\partial E_{tot}}{\partial out_{h2}} * \frac{\partial out_{h2}}{\partial y_{h2}} * \frac{\partial y_{h2}}{\partial wh_{2,2}} = -0.1262 * 0.25 * 0.25 = -0{,}007864$$

Backpropagation : exemple 20/24

Dérivées partielles pour $wh_{2,3}$:

$$\frac{\partial E_{tot}}{\partial wh_{2,3}} = \frac{\partial E_{tot}}{\partial out_{h2}} * \frac{\partial out_{h2}}{\partial y_{h2}} * \frac{\partial y_{h2}}{\partial wh_{2,3}}$$

On a déjà calculé : $\frac{\partial E_{tot}}{\partial out_{h2}} = -0.1262$ et $\frac{\partial out_{h2}}{\partial y_{h2}} = 0.25$

Il reste à calculer :

$$\frac{\partial y_{h2}}{\partial wh_{2,3}} = \frac{\partial}{\partial wh_{2,3}} \mathrm{x}_1 * wh_{2,1} + \mathrm{x}_2 * wh_{2,2} + \mathrm{x}_3 * wh_{2,3} + b_{h2} = \mathrm{x}_3 = -0.5$$

On obtient donc :

$$\frac{\partial E_{tot}}{\partial wh_{2,2}} = \frac{\partial E_{tot}}{\partial out_{h2}} * \frac{\partial out_{h2}}{\partial y_{h2}} * \frac{\partial y_{h2}}{\partial wh_{2,2}} = -0.1262 * 0.25 * -0.5 = -0{,}007864353$$

Backpropagation : exemple 21/24

Dérivées partielles pour bh_2 :

$$\frac{\partial E_{tot}}{\partial bh_2} = \frac{\partial E_{tot}}{\partial out_{h2}} * \frac{\partial out_{h2}}{\partial y_{h2}} * \frac{\partial y_{h2}}{\partial bh_2}$$

On a déjà calculé : $\frac{\partial E_{tot}}{\partial out_{h2}} = -0.1262$ *et* $\frac{\partial out_{h2}}{\partial y_{h2}} = 0.25$

Il reste :

$$\frac{\partial y_{h2}}{\partial bh_2} = \frac{\partial}{\partial bh_2} x_1 * wh_{2,1} + x_2 * wh_{2,2} + x_3 * wh_{2,3} + bh_2 = 1$$

donc

$$\frac{\partial E_{tot}}{\partial bh_2} = \frac{\partial E_{tot}}{\partial out_{h2}} * \frac{\partial out_{h2}}{\partial y_{h2}} * \frac{\partial y_{h2}}{\partial bh_2} = -0.1262 * 0.25 * 1 = -0{,}031457413$$

Backpropagation : exemple 22/24

Si on utilise SGD pour modifier les poids de h2 :

$$wh'_{2,1} = wh_{2,1} - \eta * \frac{\partial E_{tot}}{\partial wh_{2,1}} = 0.3 - 0.001 * (-0.004719) = 0{,}300004719$$

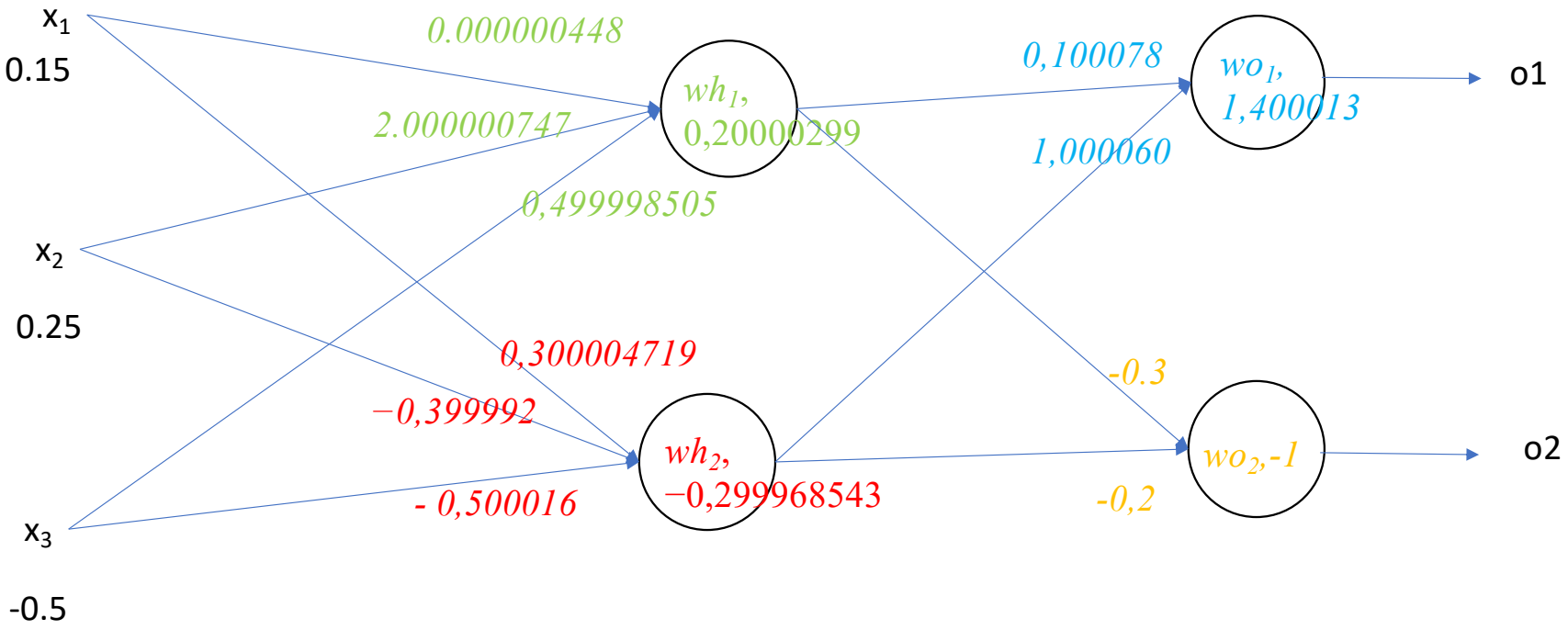
$$wh'_{2,2} = wh_{2,2} - \eta * \frac{\partial E_{tot}}{\partial wh_{2,2}} = -0.4 - 0.001 * (-0{,}00000786) = -0{,}399992$$

$$wh'_{2,3} = wh_{2,3} - \eta * \frac{\partial E_{tot}}{\partial wh_{2,2}} = -0.4 - 0.001 * (-0{,}00000786) = -0{,}500016$$

$$bh'_2 = bh_2 - \eta * \frac{\partial E_{tot}}{\partial b_{h2}} = -0{,}299968543$$

Backpropagation : exemple 23/24

Après modification de tous les poids, le réseau est



Backpropagation : exemple 24/24

Inférence avec les nouveaux poids du réseau :

out_{o1} = 0,873846655

out_{o2} = 0,217717583

Calcul de l'erreur (Cross entropy, cf slide 54) après mise à jour : $CELoss'$

Donc $P_1 =$ 0,873846655, et $P_2 =$ 0,217717583, et la vérité-terrain y_1 = 1 , et y_2 = 0

$$CELoss' = -(1 * \log(0{,}873846655) + 0 * \log(0{,}217717583))$$

$$= \mathbf{0{,}05856}$$

Cette loss $CELoss'$ est inférieure à la $CELoss$ calculée sur le réseau avec les poids initiaux (**0.05858**), donc l'apprentissage a bien fonctionné :

il a fait diminuer l'erreur sur la donnée d'apprentissage…

… mais très faiblement même sur un réseau très petit => il faut beaucoup d'exemples !!!