



# PROJET APPRENTISSAGE STATISTIQUE PROFOND

Avril 2020

***Abdoulaye KOROKO***

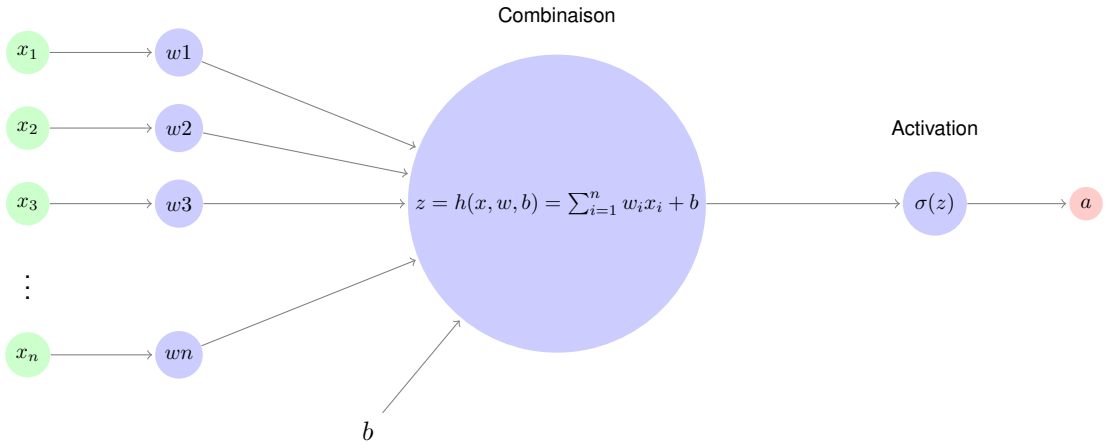
prof : ANNICK VALIBOUZE

# Table des matières

# 1 Perceptron multicouches (PMC)

## 1.1 Un perceptron ou neurone

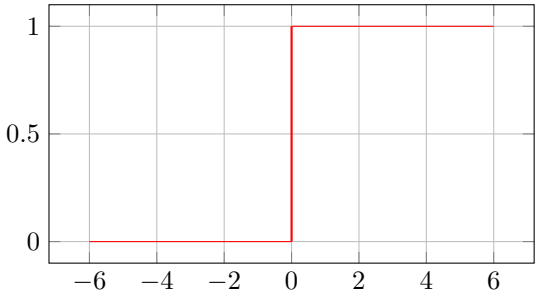
Un perception est un classifieur binaire. Il prend en entrée un vecteur $x_i \in \mathbb{R}^n$ et calcule une sortie binaire $a$ appelée activation ($a = 0$ ou $a = 1$).




Les fonctions $(x, w, b) \mapsto h(x, w, b)$ et $z \mapsto \sigma(z)$ sont respectivement appelées fonction d'entrée ou d'agrégation et fonction d'activation du neurone. L'application de la fonction d'activation $\sigma(z)$ produit la réponse du neurone. La fonction $\sigma(z)$ est différentiable et peut avoir plusieurs formes ayant un impact sur l'apprentissage du réseau. Les principales formes sont :

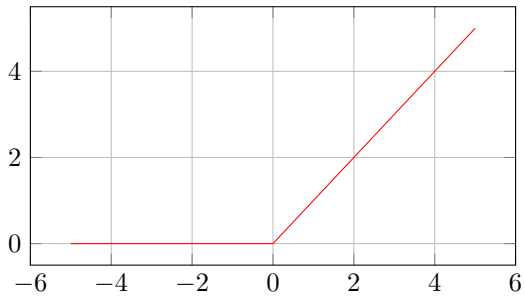
- la fonction *heaviside*,

$$\sigma(z) = \begin{cases} 0 & \text{pour} \quad z \leq 0 \\ 1 & \text{pour} \quad z > 0 \end{cases}$$



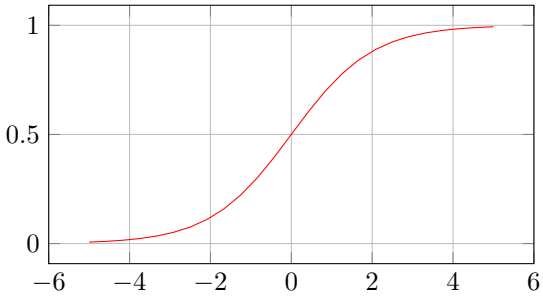

- la fonction de rectification linéaire (ou *ReLU*)

$$\sigma(z) = \begin{cases} 0 & \text{pour} \quad z \leq 0 \\ z & \text{pour} \quad z > 0 \end{cases}$$




- la fonction logistique ou *sigmoïde*,

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

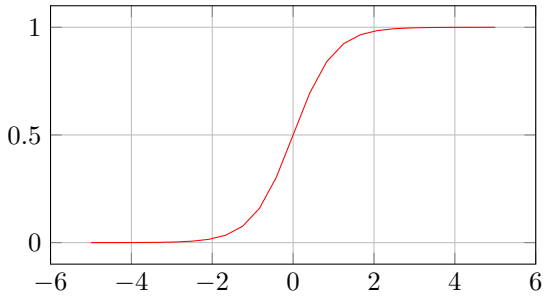



- la fonction *softmax* qui est la généralisation de la fonction logistique . Elle est utilisée dans le cas d'une classification à K classes avec $K \geq 2$. Pour un vecteur $z = (z_1, z_2, ..., z_K)$ de K nombres réels, la fonction est définie par :

$$f(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

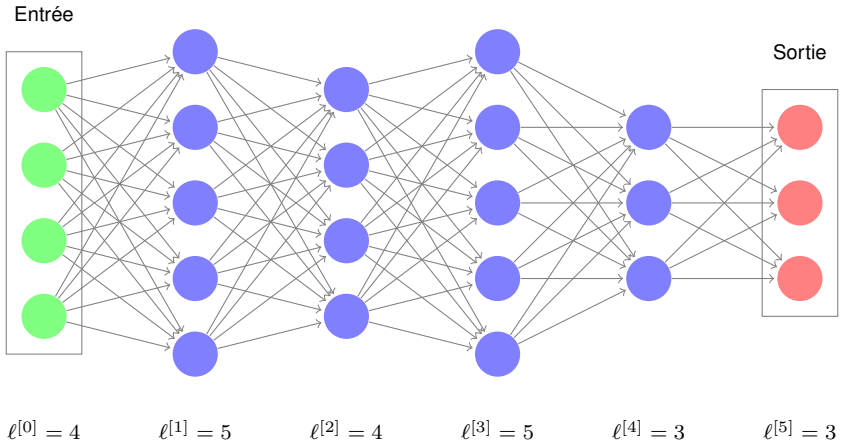
- la fonction tangente hyperbolique (ou *TanH*),

$$\sigma(z) = \tanh(z) = \frac{2}{1 + e^{-2z}} - 1$$



## 1.2 Définition du PMC

Le PMC est un réseau neuronal constitué de plusieurs couches de neurones. Un PMC contient au minimum trois couches : une couche d'entrée, une couche cachée et une couche de sortie. Ci dessous un PMC à 4 couches cachées.




Les neurones d'une couche cachée reçoivent les sorties des neurones de la couche qui précède cette couche et envoient leurs sorties aux neurones de la couche suivante à travers les poids synaptiques. De manière analogue, les neurones de la couche de sortie reçoivent en entrée les sorties de la dernière couche cachée. A part les neurones de la couche d'entrée, à chaque neurone d'une couche est associé un biais et une fonction d'activation. Par exemple pour un neurone $j$ de la couche $\ell$, $\sigma_j^{[\ell]}$ désigne sa fonction d'activation, $b_j^{[\ell]}$ son biais et $w_{j,k}^{[\ell]}$ ses poids synaptiques des connections entre ce neurone $j$ et les neurones de la couche précédente (couche $\ell - 1$). Ainsi sa réponse est $a_j^{[\ell]} = \sigma_j^{[\ell]}(z_j^{[\ell]})$ avec $z_j^{[\ell]} = \sum_k \omega_{j,k}^{[\ell]}\, a_k^{[\ell-1]} + b_j^{[\ell]}$.

## 1.3 Apprentissage des PMCs

Les PMCs font partie de la famille des algorithmes d'apprentissage supervisé qui consiste à apprendre une fonction de prédiction à partir d'une base de données labellisées. Plus formellement, l'apprentissage supervisé consiste à estimer une fonction $f$ à partir d'une base d'apprentissage $((x^{(i)}, f(x^{(i)}))_{i\in[1,n]}$ et $\forall i, x_i \in \mathbb{R}^p, p \geq 1$. Le principe d'estimation est de trouver une fonction $g$, une approximation de $f$ qui minimise l'erreur moyenne $\frac{1}{n}\sum_{i=1}^{n} L(g(x^{(i)}), f(x^{(i)}))$ où L est une fonction de coût. Un modèle d'apprentissage supervisé contient des paramètres et le processus d'apprentissage consiste à estimer les paramètres optimaux qui minimisent l'erreur moyenne. Dans le cas des PMCs, les paramètres sont les poids. Ainsi l'apprentissage du PMC consiste à résoudre le problème d'optimisation suivant :

$$\arg\min_{\omega} \frac{1}{n} \sum_{i} L(g_\omega(x^{(i)}), f(x^{(i)})) \tag{1}$$

L'apprentissage du PMC s'effectue à l'aide de l'algorithme de *backpropagation ou rétro-propagation* par utilisation de la méthode de la descente de gradient.

### 1.3.1 Algorithme de la descente de gradient

La descente de gradient est un algorithme itératif d'optimisation qui consiste à minimiser une fonction $f$ définie sur un espace $\mathbb{E}$ muni d'une norme $\|.\|$. L'algorithme permet de trouver dans le cas idéal le minimum global et dans le cas défavorable un minimum local de la fonction. Plusieurs versions de cet algorithme ont été mises en place afin d'éviter les problèmes de minimas locaux. On part d'un point initial $x_0 \in \mathbb{E}$ et on répète jusqu'à convergence (selon critère d'arrêt) la procédure suivante [fig.1] :

1. Calcul de $\nabla f(x_k)$ le gradient de $f$ en $x_k$
2. Calcul du nouvel itéré $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k \nabla f(\boldsymbol{x}_k)$ où $\alpha_k$ désigne le pas.
3. Arrêt si $\|\nabla f(x_k)\| \leqslant \varepsilon$, sinon revenir à 1 (critère de convergence).

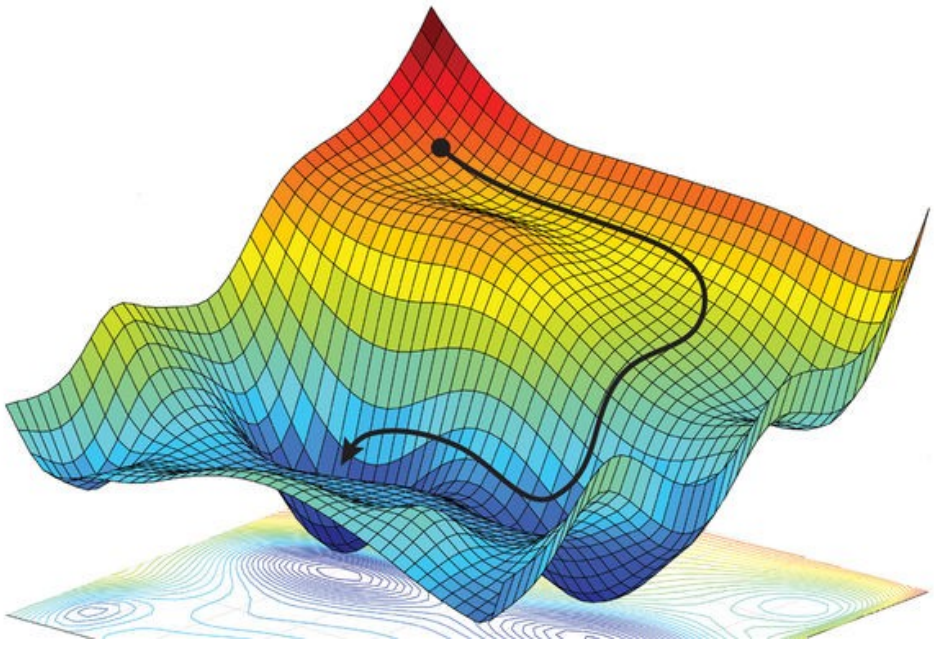


FIGURE 1: Algorithme de descente de gradient

### 1.3.2 Algorithme de rétro-propagation

La rétro-propagation est un algorithme de calcul de gradient de l'erreur et de mise à jour des poids de chaque neurone du PMC en partant de la couche de sortie jusqu'à la première couche. Considérons un PMC à $L$ couches cachées. Designons par $i$ et $o$ les couches d'entrée et de sortie respectivement. Supposons que l'on veuille entraîner le PMC avec un corpus composé d'exemples $(x^i, d^i)_{i\in\{1,..,n\}}$ avec $(x^i, d^i) \in \mathbb{R}^p \times \mathbb{R}^{N_o} \ \ \forall i$. On initialise tous les poids et les biais par des valeurs aléatoires. On calcule une prédiction $y^i$ de $x^i \ \forall i$ et une mise à jour des poids de chaque neurone du réseau est effectué afin de minimiser l'erreur moyenne. Considérons un neurone $j$ d'une couche $\ell$ du PMC. Pour un exemple $(x, d)$ du corpus, la mise à jour de ses poids s'effectue par :

$$w_{jk}^{[\ell]} = w_{jk}^{[\ell]} + \Delta w_{jk}^{[\ell]} \tag{2}$$

avec

$$\Delta w_{jk}^{[\ell]} = -\gamma \frac{\partial E}{\partial w_{jk}^{[\ell]}}$$

où $k$ désigne un neurone de la couche $\ell - 1$ et $\gamma$ le pas d'apprentissage.

$$E = \frac{1}{2} \sum_{j=1}^{N_o} (d_j - y_j)^2 = \frac{1}{2} \sum_{j=1}^{N_o} \left( d_j - a_j^{[o]} \right)^2 \tag{3}$$

désigne l'erreur quadratique moyenne ($N_o$ désigne le nombre de neurones la couche de sortie). La fonction d'erreur peut avoir une autre expression (par ex. entropie croisée). Désignons l' activation par :

$$a_j^{[\ell]} = \sigma_j^{[\ell]} \left( z_j^{[\ell]} \right) \tag{4}$$

avec $\sigma_j^{[\ell]}$ la fonction d'activation et

$$z_j^{[\ell]} = \sum_k w_{jk}^{[\ell]} a_k^{[\ell-1]} + b_j^{[\ell]} \tag{5}$$

$$\frac{\partial E}{\partial w_{jk}^{[\ell]}} = \frac{\partial E}{\partial z_j^{[\ell]}} \frac{\partial z_j^{[\ell]}}{\partial w_{jk}^{[\ell]}} \tag{6}$$

Or d'après l'équation précédente,

$$\frac{\partial z_j^{[\ell]}}{\partial w_{jk}^{[\ell]}} = a_j^{[\ell-1]} \tag{7}$$

Définissons :

$$\delta_j^{[\ell]} = -\frac{\partial E}{\partial z_j^{[\ell]}} \tag{8}$$

On a alors :

$$\Delta w_{jk}^{[\ell]} = \gamma \delta_j^{[\ell]} a_j^{[\ell-1]} \tag{9}$$

Pour calculer $\delta_j^{[\ell]}$, la règle de la chaine est appliquée :

$$\delta_j^{[\ell]} = -\frac{\partial E}{\partial z_j^{[\ell]}} = -\frac{\partial E}{\partial a_j^{[\ell]}} \frac{\partial a_j^{[\ell]}}{\partial z_j^{[\ell]}} \tag{10}$$

et

$$\frac{\partial a_j^{[\ell]}}{\partial z_j^{[\ell]}} = (\sigma_j^{[\ell]})' \left( z_j^{[\ell]} \right) \tag{11}$$

Si $\ell = o$ (couche de sortie) alors,

$$\frac{\partial E}{\partial a_j^{[o]}} = -\left( d_j - a_j^{[o]} \right) \tag{12}$$

et on obtient pour tout neurone $j$ de la couche de sortie :

$$\delta_j^{[o]} = \left(d_j - a_j^{[o]}\right)\left(\sigma_j^{[o]}\right)'\left(z_j^{[o]}\right) \tag{13}$$

Si $\ell$ est une couche cachée, alors par la règle de la chaîne :

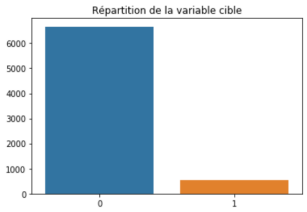
$$\begin{aligned}
\frac{\partial E}{\partial a_j^{[\ell]}} &= \sum_{k=1}^{N_{[\ell+1]}} \frac{\partial E}{\partial z_k^{[\ell+1]}} \frac{\partial z_k^{[\ell+1]}}{\partial a_j^{[\ell]}} \\
&= \sum_{k=1}^{N_{[\ell]+1}} \frac{\partial E}{\partial z_k^{[\ell+1]}} \frac{\partial}{\partial a_j^{[\ell]}} \sum_{p=1}^{N_{[\ell]}} w_{pk}^{[\ell+1]} a_p^{[\ell]} \\
&= \sum_{k=1}^{N_{[\ell+1]}} \frac{\partial E}{\partial z_k^{[\ell+1]}} w_{kj}^{[\ell+1]} a_j^{[\ell]} \\
&= -\sum_{k=1}^{N_{[\ell+1]}} \delta_k^{[\ell+1]} w_{kj}^{[\ell+1]} a_j^{[\ell]}
\end{aligned} \tag{14}$$

La réitération de la relation de récurrence (14) jusqu'à la couche de sortie permet de mettre à jour les poids de n'importe quel neurone du PMC.

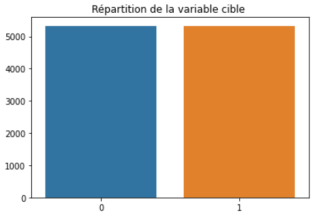
## 1.4 Application : détection d'hypothyroïdie

### 1.4.1 Contexte et base de données

L'hypothyroïdie est une insuffisance de production d'hormones par la glande thyroïde. L'objectif de cette section est de pouvoir à travers un PMC et des données sur une personne prédire si la personne a l'hypothyroïdie ou si elle est normale. Nous disposons d'une base de données labialisées [7] qui nous permettra d'entraîner et tester notre modèle de réseau. La base de données contient 6 variables explicatives et une étiquette $y$ ($y = 0$ pour les sujets sains et $y = 1$ pour les malades) pour chaque exemple. Faisant face à un problème de données fortement déséquilibrées (beaucoup plus de sains que de malades), nous avons appliqué une technique d'augmentation de données (augmenter la classe minoritaire) afin d'éviter le sur-apprentissage [fig.2].




(a) Données originales




(b) données augmentées

FIGURE 2: Augmentation de données

### 1.4.2 Construction et apprentissage du réseau

Le réseau a été construit à l'aide du framework deep learning `pytorch` [8] en `python`. Nous avons construit un petit PMC à deux couches cachées. Nous avons entraîné ce PMC pendant 500 époques. Nous avons pu observer l'évolution les erreurs sur les données d'apprentissage et de test [fig.3].

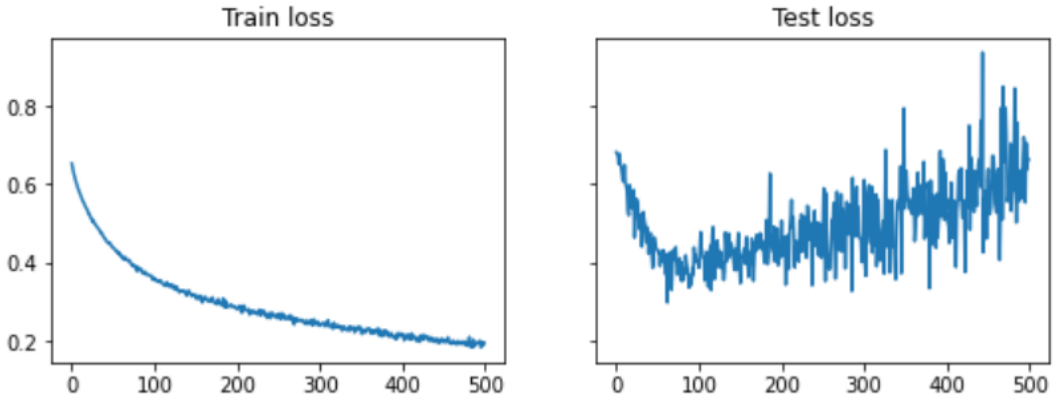



FIGURE 3: Evolution des erreurs pendant l'apprentissage du PMC

### 1.4.3 Inférence

Nous avons évalué notre modèle à l'aide de deux métriques : l'*accuracy* et le *ROC score*. La première métrique évalue la capacité du modèle à prédire de résultas vrais tandis que la seconde tient compte des vrais positifs, vrais négatifs, faux positifs et faux négatifs. Voir l'article [9] pour la définition du *ROC score*.

$$accuracy = \frac{nombre\ de\ prédiction\ vraies}{nombre\ total\ de\ données}$$

Sur une donnée de test de taille 1440 avons obtenu un *accuracy* de $93\%$ et un *ROC score* de $73\%$. Par ailleurs, nous avons comparé notre réseau de neurones à d'autres méthodes d'apprentissage statistique [fig.4].

| | classifiers | accuracy | roc score |
|---|---|---|---|
| 0 | RandomForestClassifier | 0.985417 | 0.983360 |
| 1 | GradientBoostingClassifier | 0.987500 | 0.988871 |
| 2 | AdaBoostClassifier | 0.986111 | 0.988122 |
| 3 | DecisionTreeClassifier | 0.983333 | 0.947138 |
| 4 | KNeighborsClassifier | 0.879861 | 0.746549 |
| 5 | GaussianProcessClassifier | 0.871528 | 0.812253 |
| 6 | PMC | 0.933333 | 0.731514 |

FIGURE 4: Comparaison du PMC avec d'autres méthodes de machine learning

# 2 Les réseaux de neurones convolutifs (CNNs)

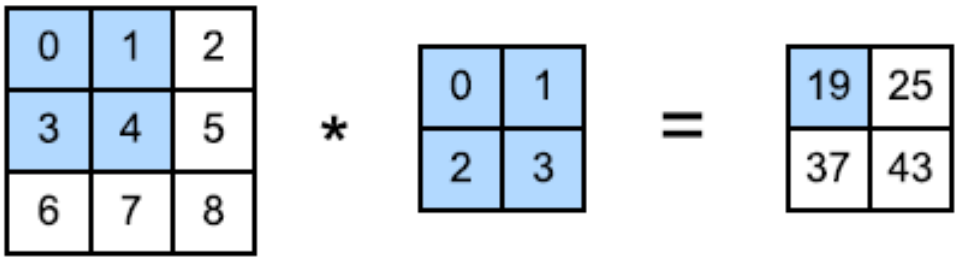
## 2.1 généralités

Les perceptrons multi-couches fonctionnent bien lorsqu'on travaille avec des données structurées. Nous appelons données structurées des données tabulées où chaque ligne correspond à un individu ou un exemple dans l'ensemble des données d'entraînement et chaque colonne une variable explicative (une caractéristique ou un feature). Ces réseaux performent moins bien en terme de précision et de temps de calcul quand il s'agit des données non structurées comme des images ou des vidéos. Par exemple, pour une simple tâche de classification binaire d'images (chien/chat) avec un réseau de neurones artificiels, l'on peut se retrouver facilement avec un modèle à $10^9$ paramètres. A moins d'avoir des mégadonnées (de l'ordre de milliards), assez de GPUs, des méthodes robustes et efficaces d'optimisation et une patience extraordinaire, l'apprentissage des paramètres de ce modèle s'avère impossible. Pour pallier ce problème, les réseaux de neurones convolutifs basés sur le principe de la convolution mathématique ont été introduits.

## 2.2 Fondements des CNNs

### 2.2.1 Produit de convolutions

En mathématiques, la convolution entre deux fonctions $f, g : \mathbb{R}^d \to \mathbb{R}$ est définie par :
$f(x) \circledast g(x) = \int_{\mathbb{R}^d} f(z)g(x-z)dz$
Dans le cas d'une mesure discrète (par exemple la mesure de comptage sur $\mathbb{Z}$), la convolution se définit par $f(i) \circledast g(i) = \sum_{a\in\mathbb{Z}} f(a)g(i-a)$.
Dans le cas de deux matrices $X \in \mathbb{M}_{nx,px}$ et $F \in \mathbb{M}_{nf,pf}$, la convolution entre $X$ et $F$ donne une matrice $R \in \mathbb{M}_{nr,pr}$ avec $nr = nx - nf + 1$ et $pr = px - pf + 1$.
$\forall (i,j) \in \{0,..,nr\} \times \{0,..,pr\}, R(i,j) = \sum_{n=0}^{nx-1} \sum_{p=0}^{px-1} X(n,p)F(i-n, j-p)$.
Une image numérique étant représentée par une matrice de pixels (2D pour une image blanc-noir et 3D pour une image en couleur), la convolution entre deux matrices est fréquemment utilisée pour le traitement d'image, comme le lissage, la netteté et la détection des bords des images. L'entrée $X$ désigne l'image et $F$ le filtre ou noyau. Un simple exemple de convolution entre deux matrices :




On peut décider avec un choix adéquat du filtre augmenter le contraste, rendre flou ou augmenter les bords d'une image.

### 2.2.2 Options de convolutions

• **Padding (extension des bords de l'image)**

Le produit de convolution d'une image de taille $n \times p$ par un filtre de taille $f \times f$ donne une image de taille $(n - f + 1) \times (p - f + 1)$. Ce résultat cause deux principaux problèmes :

-Rétrécissement de l'image

-Perte d'informations qui sont sur les bords de l'image

Pour résoudre ces deux soucis, on peut utiliser la technique du padding qui consiste à étendre les bords de l'image (ajout de lignes et de colonnes) avant d'appliquer la convolution. Généralement, l'on met des zéros aux lignes et colonnes additionnelles. Si l'on convole une image de taille $n \times n$ par un noyau de taille $f \times f$ avec un padding $p$, on obtient une image de taille $(n + 2p - f + 1) \times (n + 2p - f + 1)$.

Le padding s'avère nécessaire lorsqu'on utilise des réseaux à convolutions très profonds.

• **Strided convolution (Convolution avec saut de pas)**

La convolution avec saut de pas consiste à sauter certains pixels de l'image pendant le produit de convolution. Lorsqu'on effectue la convolution d'une image de taille $n \times n$ par un un filtre de taille $f \times f$ avec un padding $p$ et un pas $s$, on obtient une image de taille $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$.

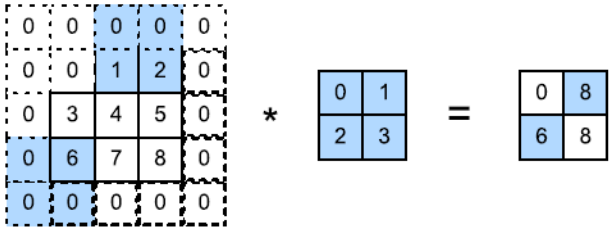
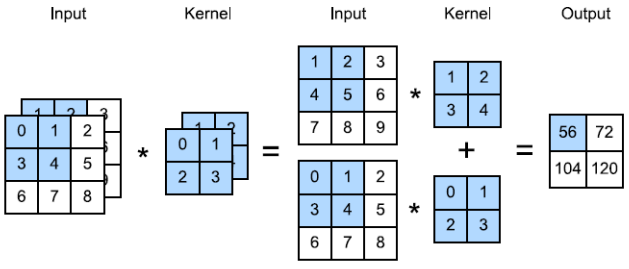



FIGURE 5: Convolution avec un padding $p = 1$ et pas égaux à 3 et 2 pour la ligne et la colonne respectivement

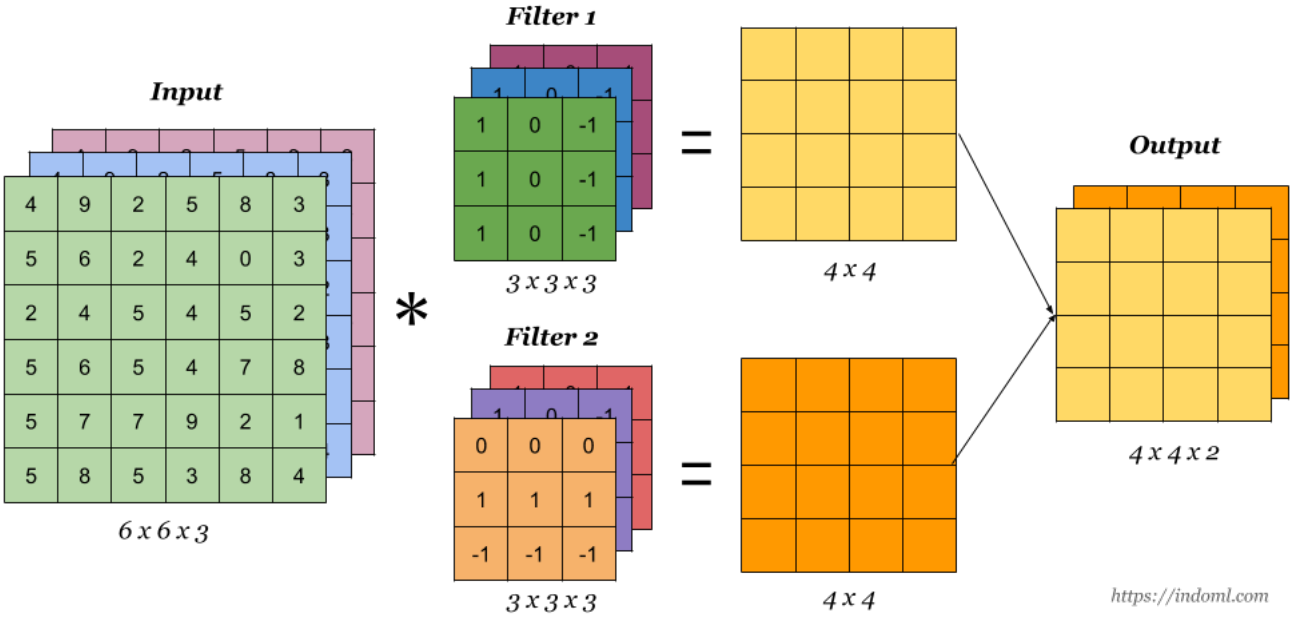
### 2.2.3 Convolutions 3D

La convolution 3D s'effectue avec une matrice de taille $h \times w \times c$ et un filtre de taille $f \times f \times c$. Il faut noter que les dernières dimensions que l'on appelle *channel* de la matrice et du filtre doivent être identiques (ici $c$). On obtient une matrice 2D de taille $(w - f + 1) \times (w - f + 1)$.




On peut utiliser plusieurs filtres sur la même image pour détecter plusieurs caractéristiques de l'image.

Dans ce cas le résultat de la convolution est la concaténation des résultats de convolution de l'image avec chacun des filtres. Par exemple :




### 2.2.4 Une couche de convolution dans un réseau

Etant donnée une entrée X :

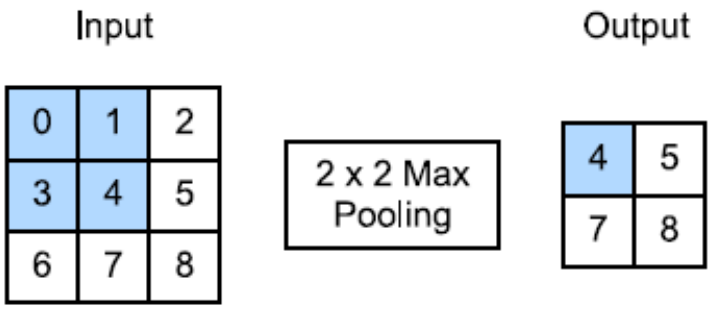
- Appliquer un padding $p$ (optionnel)
- Choisir un pas $s$
- Choisir la dimension et le nombre de filtres
- Effectuer la convolution de X avec chacun des filtres et ensuite concaténer les différents résultats pour obtenir le résultat final
- Ajouter un biais (optionnel)
- Appliquer une fonction d'activation (ReLU)

Il faut noter que les paramètres de la convolution ne dépendent pas de l'entrée (image). Elles ne dépendent que de la taille, du nombre de filtres utilisés et éventuellement du biais.

### 2.2.5 Couche de pooling

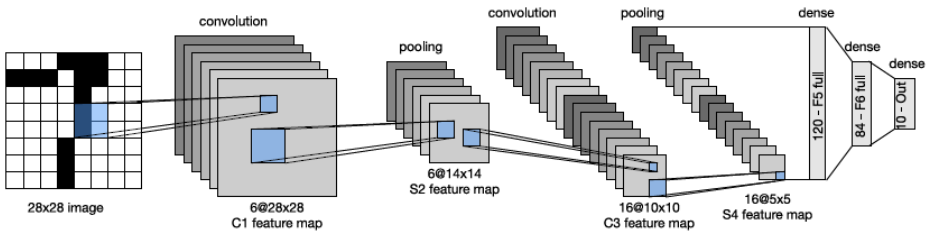
Comme les couches convolutives, les opérateurs de pooling se composent d'une fenêtre de forme (taille) fixe (appélée sliding window ou fenêtre glissante) qui est glissée sur toutes les régions de l'entrée en fonction du pas , et calcule une seule sortie pour chaque région traversée par la fenêtre. Cependant, contrairement à la couche de convolution, la couche de pooling ne contient aucun paramètre (il n'y a pas de filtre). La couche de pooling calcule généralement la valeur maximale ou moyenne des éléments dans le sliding window. Ces opérations sont appelées maximum pooling et average pooling, respectivement. Les couches de pooling se trouvent généralement juste après chaque couche de convolu-

tions. Leur fonction principale est de réduire progressivement la taille spatiale de la représentation pour réduire la quantité de paramètres tout en retenant les informations importantes.




### 2.2.6 Réseau de neurones à convolution (ConvNet)

Un ConvNet est la succession de plusieurs couches de convolutions ( éventuellement suivies de couches de pooling) suivies des couches de neurones (fully connected layers).




## 2.3 Application : Diagnostic du covid-19 à travers des images radiographiques

### 2.3.1 Contexte et données

Nous avons à notre disposition une base de données d'images radiographiques des poumons. Cette base de données [15] créée par une équipe de chercheurs et de médecins contient des images de poumons de cas normaux, de COVID-19, de pneumonies virales [fig.6].
Nous souhaitons à travers cette base de données entraîner un réseau à convolution à être capable de prédire la classe (normale, pneumonie virale ou covid-19) d'une future image radio-graphique des poumons.

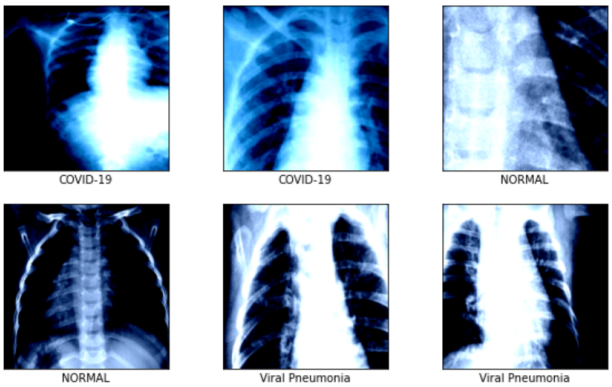



FIGURE 6: Quelques images de la base de données

### 2.3.2 Mise en place et entraînement du réseau

Le réseau a été mis en place à l'aide du framework d'apprentissage profond `pytorch` [8] en utilisant la technique du transfert learning. Nous avons utilisé deux réseaux dans le but de les comparer et rétenir celui qui a un bon score sur la base de données de test. :

- *resnet18* un réseau de neurones convolutifs profond contenant 18 couches.
- *darknet53* un réseau à convolution très profond (contient 53 couches)

La base de données d'apprentissage contient 2500 images dont 289 de covid-19, 1095 de normal et le reste de pneumonie virale [fig.7].

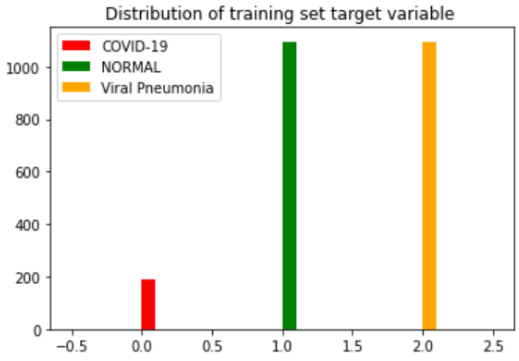
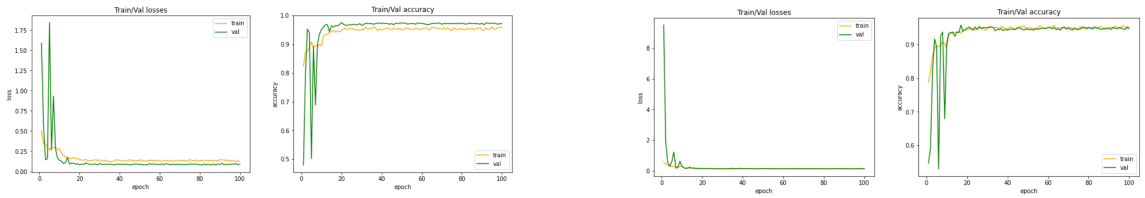



FIGURE 7: Distribution de la variable cible

Nous avons entraîné ces deux réseaux avec notre base de données d'apprentissage pendant 100 époques et en utilisant des techniques d'augmentation de données [fig.8].




(a) resnet18 (b) darknet53

FIGURE 8: Courbes d'entraînement

### 2.3.3 Inférence

Nous avons testé les deux réseaux sur une base de données de test contenant au total 530 images.

- **resnet18** Comme l'indique la figure [9], notre modèle a une sensibilité de $100\%$ au covid-19 (détecte tous les cas de covid-19), une précision de $97\%$ sur le covid-19 (rarement de faux positifs) et réalise un accuracy de $98\%$ sur l'ensemble des 3 classes.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| COVID-19 | 0.97 | 1.00 | 0.98 | 30 |
| NORMAL | 0.97 | 0.98 | 0.98 | 250 |
| Viral Pneumonia | 0.98 | 0.96 | 0.97 | 250 |
| accuracy | | | 0.98 | 530 |
| macro avg | 0.97 | 0.98 | 0.98 | 530 |
| weighted avg | 0.98 | 0.98 | 0.98 | 530 |

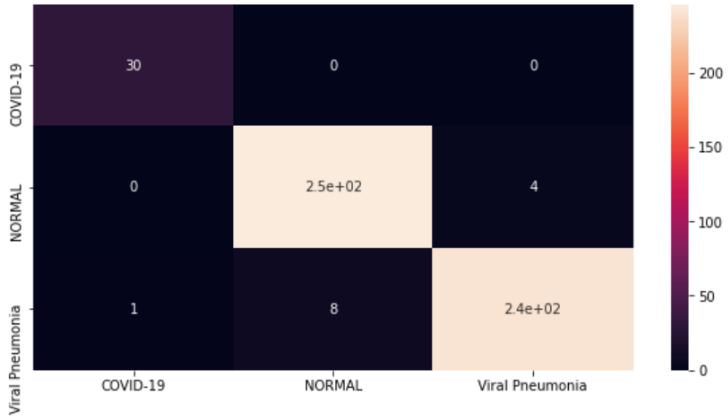



FIGURE 9: Résultats d'évaluation resnet18

- **darknet53** Moins performant que *resnet18*, il a une sensibilité et une précision de $93\%$ sur le covid-19 puis un accuracy de $96\%$ sur l'ensemble des 3 classes.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| COVID-19 | 0.93 | 0.93 | 0.93 | 30 |
| NORMAL | 0.93 | 1.00 | 0.97 | 250 |
| Viral Pneumonia | 0.99 | 0.92 | 0.95 | 250 |
| accuracy | | | 0.96 | 530 |
| macro avg | 0.95 | 0.95 | 0.95 | 530 |
| weighted avg | 0.96 | 0.96 | 0.96 | 530 |

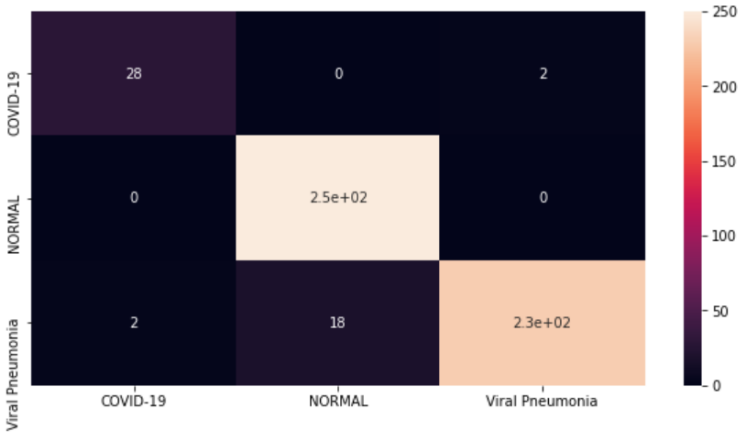



FIGURE 10: Résultats d'évaluation darknet53

*darknet53* est moins perfomant que *resnet18* car étant très profond, il nécessite beaucoup plus de données d'apprentissage afin d'éviter le sur-apprentissage.

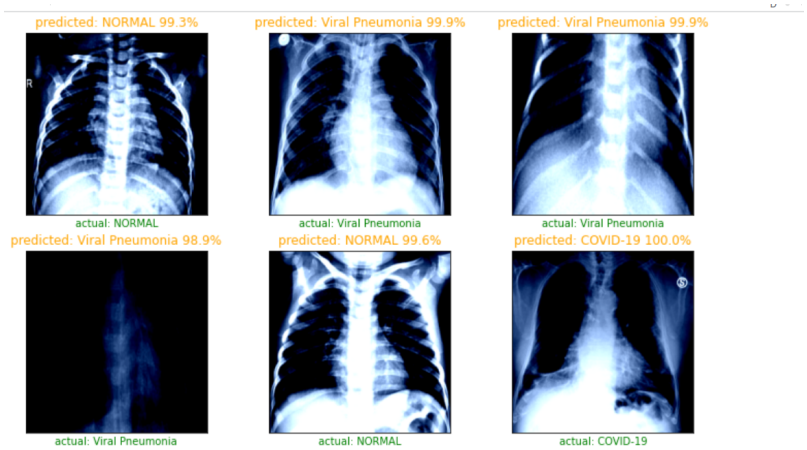



FIGURE 11: Quelques prédictions avec resnet18

# 3 Cartes auto-adaptatives de Kohonen

## 3.1 Définition

Les cartes auto-adaptatives de Kohonen ou encore en anglais Self-organazing-Maps (MAPs) ont été inventées dans les années 1980 par Teuvo Kohonen et désignent un algorithme d'apprentissage profond non supervisé. Une SOM est un type de réseau de neurones artificiels qui n'a pas de couches cachées et crée une représentation (une carte) des données d'entrée à travers sa couche de sortie [fig.12]. Il s'agit typiquement d'un algorithme de réduction de dimension. Les SOMs peuvent être utilisées pour la visualisation de données à grandes dimensions ou pour la classification non supervisée (clustering).

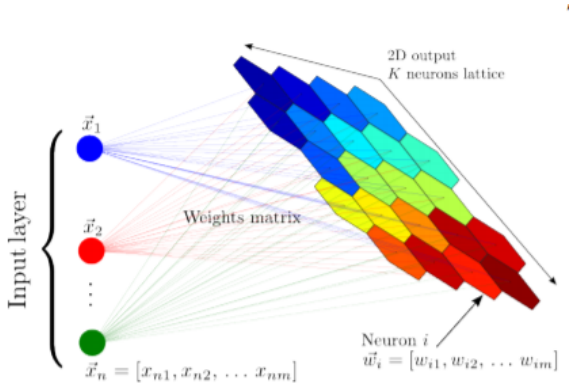



FIGURE 12: Réduction de dimension avec les SOMs : la base de données d'apprentissage composée de n exemples multidimensionnels ($x_i \in \mathbb{R}^m \ \forall i \in \{1, .., n\}$) est projetée sur une carte 2D. Les projections des exemples ayant des propriétés similaires sont proches sur la carte 2D [4].

## 3.2 Algorithme d'apprentissage d'une SOM

Considérons une SOM à K neurones sur la carte de sortie et un jeu de données d'apprentissage à n exemples ($x_1, x_2, .., x_n$ avec $x_i = (x_{i1}, ..., x_{im}) \in \mathbb{R}^m \ \forall i \in \{1, .., n\}$). A chaque neurone $k$ de la carte de sortie est attribué un vecteur poids $w_k = (w_{k1}, ..., w_{km}) \in \mathbb{R}^m$. Ces poids sont aléatoirement initiés.
A chaque itération t, tous les exemples de la base d'apprentissage sont traités indépendamment et les poids des neurones de sortie sont mis à jour dans le but d'obtenir une représentation optimale des données d'entrée. Cette procédure conserve la topologie de l'espace des données d'apprentissage.

**Algorithme** [4]

Pour t=1,...,nombre d'itération :

Pour i=1,...,nombre d'exemples dans la base d'apprentissage :

- Calculer $d_k = d(x_i, w_k(t)) = \sqrt{\sum_{p=1}^{m}(x_{ip} - w_{kp}(t))^2} \quad \forall k \in \{1, ..., K\}$
- Prendre le numéro $b = \arg\min_{k\in\{1,...,K\}} d_k$. C'est le numéro du neurone dit gagnant c'est à dire le plus proche au sens de la distance euclidienne à la donné $x_i$
- Faire la mise à jour de tous les poids selon la formule
  $w_k(t+1) = w_k(t) + \alpha(t)H_{b,k}(t)[x_i - w_k(t)] \; \forall k \in \{1, ..., K\}$ Avec :
  $\alpha(t)$ le taux d'apprentissage (décroît à chaque itération) ;
  $H_{b,k}(t)$ la fonction de voisinage (décroît en fonction de la distance entre le neurone $b$ et $k$ et aussi en fonction de l'itération $t$). Cette fonction implique que les neurones proches du neurone gagnant ont leur poids fortement modifiés et inversement les plus éloignés ont leur poids faiblement modifiés. Le neurone gagnant essaie de regrouper tous les neurones proches de lui vers la donné $x_i$ [fig.13]

Fin Pour

Fin Pour

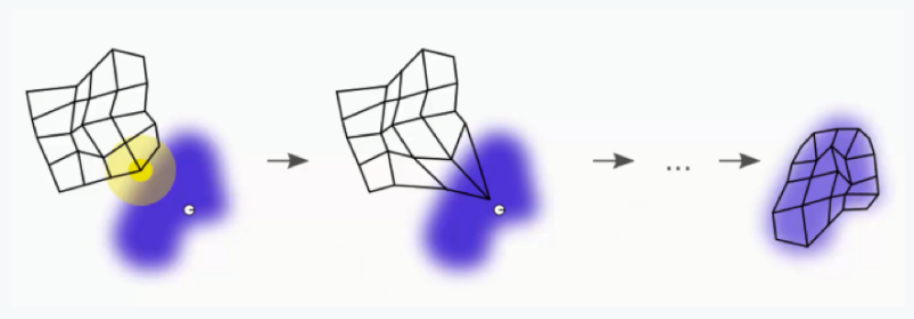


FIGURE 13: Apprentissage d'une SOM avec un exemple $x_i$ de base d'apprentissage. Le neurone gagnant tire la carte vers la donnée $x_i$ [18]

## 3.3 Application : Clustering de pays en fonction des indicateurs

### 3.3.1 Base de données et contexte

Nous disposons d'une base de données qui provient de *kaggle* [5]. Cette base de données contient 227 pays avec 20 indicateurs mesurés pour chacun des pays. Ces indicateurs sont par exemple le nombre de la population, la densité, le taux de mortalité infantile,taux de migration, etc [fig.14].
On souhaite à l'aide des cartes auto-adaptatives former des groupes des pays en fonction du niveau de vie des populations (les pays ayant approximativement le même niveau de vie devraient appartenir au même groupe).

| Country | Region | Population | Area (sq. mi.) | Pop. Density (per sq. mi.) | Coastline (coast/area ratio) | Net migration | Infant mortality (per 1000 births) | GDP ($ per capita) | Literacy (%) | Phones (per 1000) | Arable (%) | Crops (%) | Other (%) | Climate | Birthrate | Deathrate | Agriculture | I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Afghanistan | ASIA (EX. NEAR EAST) | 31056997 | 647500 | 48,0 | 0,00 | 23,06 | 163,07 | 700.0 | 36,0 | 3,2 | 12,13 | 0,22 | 87,65 | 1 | 46,6 | 20,34 | 0,38 | |
| Albania | EASTERN EUROPE | 3581655 | 28748 | 124,6 | 1,26 | -4,93 | 21,52 | 4500.0 | 86,5 | 71,2 | 21,09 | 4,42 | 74,49 | 3 | 15,11 | 5,22 | 0,232 | |
| Algeria | NORTHERN AFRICA | 32930091 | 2381740 | 13,8 | 0,04 | -0,39 | 31 | 6000.0 | 70,0 | 78,1 | 3,22 | 0,25 | 96,53 | 1 | 17,14 | 4,61 | 0,101 | |
| American Samoa | OCEANIA | 57794 | 199 | 290,4 | 58,29 | -20,71 | 9,27 | 8000.0 | 97,0 | 259,5 | 10 | 15 | 75 | 2 | 22,46 | 3,27 | NaN | |
| Andorra | WESTERN EUROPE | 71201 | 468 | 152,1 | 0,00 | 6,6 | 4,05 | 19000.0 | 100,0 | 497,2 | 2,22 | 0 | 97,78 | 3 | 8,71 | 6,25 | NaN | |

FIGURE 14: Base de données des pays

### 3.3.2 Implémentation de la carte et résultats

Nous avons utilisé la librairie open source `MiniSom` [10] pour la mise en place et l'entraînement de la carte. Nous avons décidé de former 20 groupes de pays. Pour cela nous avons utilisé une carte auto-adaptative de taille $5 \times 4$. Nous avons donc 20 neurones de sortie. Les pays ayant le même neurone de sortie gagnant forme un même groupe [fig.15]. Un neurone sur la carte représente ainsi un groupe de pays ayant à peu près le même niveau de vie.

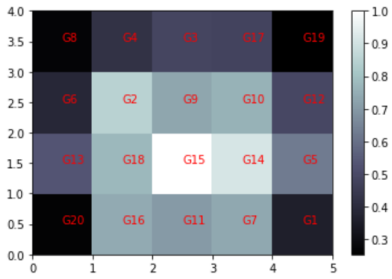



FIGURE 15: Carte des 20 neurones gagnants

Maintenant il reste à déterminer la liste des pays derrière chaque neurone sur la carte pour former notre clustering. Nous obtenons les groupes ci-dessous :

```
{'G1':  ['Afghanistan ','Benin ','Burkina Faso ','Burma ', 'Cambodia ',
            'Cameroon ','Central African Rep. ','Chad ','Congo, Dem. Rep. ',
"Cote d'Ivoire ",'Ethiopia ','Guinea-Bissau ','Laos ','Lesotho ','Liberia ',
            'Malawi ','Mali '],


'G2':  ['Albania ','Andorra ','Argentina ','Armenia ','Belarus ',
  'Bosnia & Herzegovina ','Cape Verde ','Chile ', 'Cyprus ','Estonia ',
  'Georgia ','Kazakhstan '],
```

```
'G3':  ['Algeria ','Anguilla ','Australia ','Azerbaijan ','Bahrain ',
 'Belize ','Bolivia ','Brazil ','Brunei ','Colombia ','Costa Rica ',
 'Ecuador ','French Guiana ','French Polynesia ','Greenland ','Guyana '],


'G4':  ['American Samoa ','Antigua & Barbuda ','Bahamas, The ','Barbados ',
 'British Virgin Is. ', 'Cook Islands ', 'Croatia ', 'Cuba ', 'Fiji ',
 'Guadeloupe ', 'Guam '],


'G5': ['Angola ','Bhutan ','Botswana ','Congo, Repub. of the ','Djibouti ',
 'Equatorial Guinea ','Gabon ','Guinea ','Iraq ','Mozambique ','Niger ',
 'Sierra Leone ','Swaziland '],


'G6': ['Aruba ','Austria ','Belgium ','Bermuda ','Canada ','Cayman Islands ',
 'Finland ', 'Greece ', 'Guernsey ', 'Iceland ', 'Ireland ', 'Isle of Man ',
 'Israel ','Japan ','Jersey ','Korea, South ','Luxembourg '],


'G7': ['Bangladesh ','Burundi ','Comoros ','Gambia, The ','Haiti ','India ',
 'Nigeria ','Pakistan ','Rwanda ','Togo ','Uganda '],


'G8': ['Bulgaria ','Czech Republic ','Denmark ','France ','Germany ',
 'Italy ','Latvia ','Liechtenstein ','Malta ','Netherlands ','Portugal ',
 'Slovakia ','Spain ','United Kingdom ','United States '],

'G9': ['China ','Dominica ','Dominican Republic ','Indonesia ',
 'Jamaica ','Malaysia ','Mexico '],


'G10': ['East Timor ','Egypt ','Honduras ','Kyrgyzstan ','Libya ',
 'Morocco ','Nauru ','Nicaragua ','Oman ','Panama ','Paraguay ','Tuvalu '],


'G11': ['El Salvador ','Gaza Strip ', 'Kiribati ','Marshall Islands ',
 'Micronesia, Fed. St. ','Sao Tome & Principe ','Tonga '],

'G12': ['Eritrea ','Madagascar ','Mauritania ','Nepal ',
 'Papua New Guinea ','Senegal ','Somalia ','Sudan ','Tanzania ',
 'Vanuatu ','Yemen ','Zambia '],


'G13': ['Faroe Islands ','Gibraltar ', 'Hong Kong ', 'Macau ',
```

```
  'Martinique ','Monaco ','Netherlands Antilles ','New Caledonia ',
  'New Zealand ','Norway ','Palau ','San Marino ','Singapore ',
  'Slovenia ','Sweden ','Switzerland '],


 'G14': ['Ghana ', 'Guatemala ', 'Kenya ', 'Maldives ', 'Mayotte '],


 'G15': ['Grenada ','Philippines ','Saint Lucia ',
'Saint Vincent and the Grenadines ','Samoa ','Sri Lanka ',
  'Tunisia ','Wallis and Futuna ','West Bank '],


 'G16': ['Hungary ','Lithuania ','Mauritius ','Moldova ',
  'Poland ','Romania ','Serbia ','Thailand ','Turkey ','Ukraine '],


 'G17': ['Iran ','Jordan ','Kuwait ','Peru ','Puerto Rico ',
  'Qatar ','Saudi Arabia ','Suriname ','Turks & Caicos Is ',
  'United Arab Emirates ','Venezuela ', 'Western Sahara '],


 'G18': ['Korea, North ','Lebanon ','Macedonia ','Montserrat ',
  'N. Mariana Islands '],

 'G19': ['Mongolia ','Namibia ','Solomon Islands ','South Africa ',
  'Syria ','Tajikistan ','Turkmenistan ','Uzbekistan ','Zimbabwe '],


 'G20': ['Reunion ','Russia ','Saint Helena ','Saint Kitts & Nevis ',
  'St Pierre & Miquelon ','Seychelles ','Taiwan ','Trinidad & Tobago ',
              'Uruguay ','Vietnam ', 'Virgin Islands ']}
```

Nous constatons que les pays sont bien regroupés par niveau de niveau. Par ailleurs les groupes proches sur la carte ont des niveaux de vie proches.

# 4 Apprentissage par renforcement

## 4.1 Définition

L'apprentissage par renforcement est une branche d'intelligence artificielle qui consiste à entraîner un agent qui interagit avec un environnement de façon à le rendre autonome. L'argent arrive dans des différents états de l'environnement en prenant des actions. À chaque action est associée une récompense qui peut être positive ou négative. Le processus d'apprentissage consiste à apprendre a l'agent une stratégique qui lui permettra d'amasser le maximum possible de récompense positive.

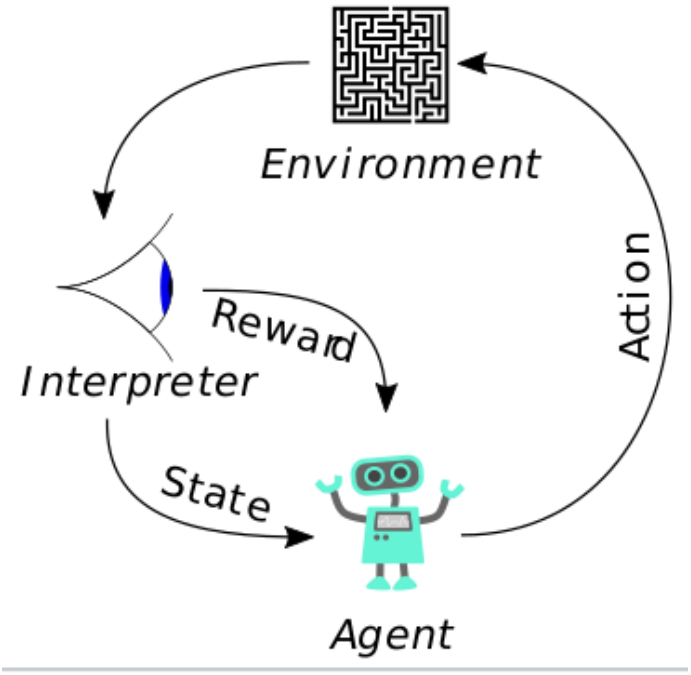



FIGURE 16: Interactions agent-environnement [20]

## 4.2 Applications

L'apprentissage par renforcement est utilisé dans plusieurs domaines :

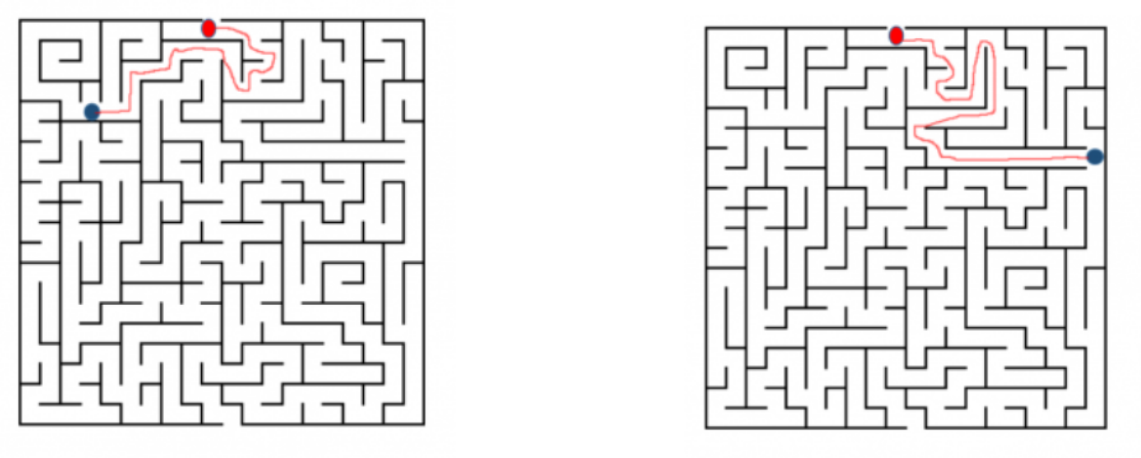
- *Systèmes de recommandations* : Suggérer des produits, informations, services, etc selon les préférences des utilisateurs.
- *Energie* : Prendre des décisions adaptées pour une consommation efficace de l'énergie.
- *Finance* : pricing, trading, gestion de risque,optimisation des portfolio.
- *Santé* : diagnostics
- *Robotique*
- *Transport* : conduite autonome, gestion flux de commandes de billet, etc
- *Jeux vidéos* Alpha Go

## 4.3 Processus de décision Markoviens

### 4.3.1 Préambule

Chaque état dans l'environnement est la conséquence de l'état qui le précède qui est lui même la conséquence d'un état antérieur. Stocker ces informations mêmes avec un environnement de nombre petit d'états est pratiquement infaisable.

Pour résoudre ce problème, une hypothèse est émise : chaque état suit la propriété de Markov. Autrement dit chaque état dépend uniquement de l'état qui le précède et de la transition entre cet état et l'état actuel.



(a) Chemin 1 (b) chemin 2

FIGURE 17: Illustration de la propriété de Markov [17]

Sur la [Fig.17], dans chacune des deux scènes, le point bleu est le point de départ de l'argent et le point rouge son point d'arrivée. L'argent débute au point bleu et emprunte différents chemins pour atteindre le point rouge. Le chemin emprunté pour arriver au point rouge n'importe pas. La prochaine étape consiste à prendre une action pour sortir du labyrinthe. Clairement, dans chacun des deux cas, l'on a besoin uniquement de l'information dans l'état rouge pour prendre la bonne action permettant de réussir le jeu et c'est ce que la propriété de Markov implique.

### 4.3.2 Définition formelle

Un processus de décision markovien est un quadruplet $(S, A, T, R)$ où $S$ et $A$ désignent un ensemble fini d'états et d'actions respectivement. $T$ la fonction de transition définie par $T : S \times A \times S \to [0; 1]$, c'est à dire la probabilité d'être à l'état $s'$ après avoir pris l'action $a$ à l'état $s$ noté $T(s, a, s')$. $R$ la fonction de récompense $R : S \times A \times S \to \mathbb{R}$.
La propriété de Markov se traduit par :

$$\mathbb{P}\left(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots\right) = \mathbb{P}\left(s_{t+1}|s_t, a_t\right) = T\left(s_t, a_t, s_{t+1}\right) \tag{15}$$

## 4.4 Stratétigue, équation de Bellman et Q-learning

Etant donné un processus de décision markovien $(S, A, T, R)$ , une stratégie ou politique est une fonction $\pi$ qui étant donné un état $s$ calcule une action $a$ associée à cet état. La politique peut être déterministe $\pi : S \to A$ ou stochastique $\pi : S \times A \to [0, 1]$. Dans la suite, nous nous placerons dans le cas déterministe. La politique est choisie conformément à la fonction de récompense R. Désignons par $r_t = R(s_t, \pi(s_t), s_{t+1})$ la récompense obtenu par l'agent après avoir opéré l'action $\pi(s_t)$ selon la politique $\pi$. L'agent peut chercher à maximiser plusieurs critères d'intérêts :

- $E(\sum_{t=0}^{h} r_t)$ espérance de la somme des récompenses à un horizon fini fixé $h$ ;
- $\liminf_{h\to+\infty} E\left(\frac{1}{h}\sum_{t=0}^{h} r_t\right)$ ou $\limsup_{h\to+\infty} E\left(\frac{1}{h}\sum_{t=0}^{h} r_t\right)$ récompense moyenne à long terme ;

- $E\left(\sum_{t=0}^{\infty}\gamma^t r_t\right)$ récompense amortie à horizon infini où $\gamma \in [0;1]$

Dans la suite, nous ne considérons que le dernier critère. Après le choix de la stratégie et le critère, deux fonctions peuvent être définies :

- $V^\pi : S \to \mathbb{R}$ fonction des valeurs des états ; $V^\pi(s)$ est le gain obtenu par l'agent s'il applique la politique $\pi$ à l'état $s$.
- $Q^\pi : S \times A \to \mathbb{R}$ c'est la fonction de valeur des états-actions, $Q^\pi(s,a)$ désigne la gain obtenu par l'argent s'il démarre à l'état $s$ et commence appliquer l'action $a$ avant d'appliquer la politique $\pi$.

On a toujours $V^\pi(s) = Q^\pi(s, \pi(s))$.
$Q^\pi(s,a) = E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k} | s_t = s, a_t = a\right\} = \sum_{s' \in S}\left[R\left(s,a,s'\right) + \gamma V^\pi\left(s'\right)\right]T\left(s,a,s'\right)$
ou encore
$V^\pi(s) = E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k} | s_t = s\right\} = \sum_{s' \in S}\left[R\left(s,\pi(s),s'\right) + \gamma V^\pi\left(s'\right)\right]T\left(s,\pi(s),s'\right)$.
Ces deux dernières équations sont connues sous le nom de l'équation de Bellman. Dans le cas où l'on cherche travaille avec la fonction des états-actions, on parle de Q-learning. Dans la suite nous ne considérons que le Q-learning. Le but de l'argent est de trouver la politique optimale $\pi^*$ qui lui permet de maximiser son gain c'est à dire $\forall$ état $s$, $\forall$ autre politique $\pi$, $Q^{\pi^*}(s,a) \geq Q^\pi(s,a)$.
La fonction de qualité optimale vérifie l'équation d'optimalité de Bellman :
$Q^*(s,a) = \sum_{s' \in S}\left[R\left(s,a,s'\right) + \gamma \max_{a' \in A} Q^*\left(s',a'\right)\right]T\left(s,a,s'\right)$.
Le Q-learning consite à apprendre les valeurs $Q^*(s,a) \quad \forall (s,a) \in S \times A$. [fig.18]

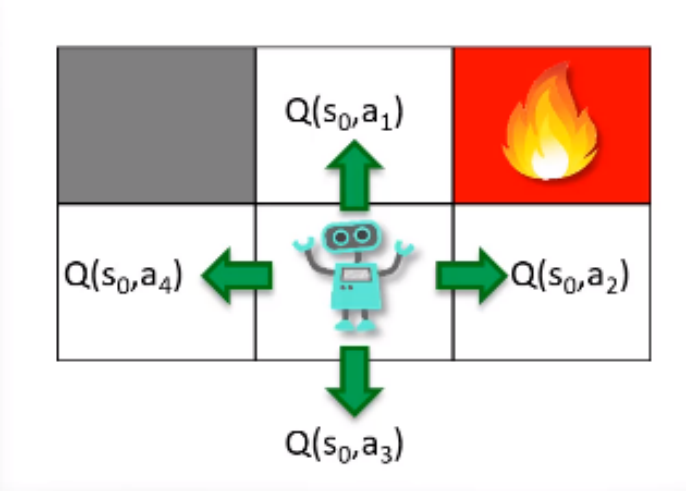



FIGURE 18: les valeurs Q(s,a) [19]

La mise à jour des valeurs $Q^*(s,a)$ se fait à travers l'algorithme :

$$Q_t^*(s,a) = Q_{t-1}^*(s,a) + \alpha TD_t(s,a) \tag{16}$$

où

$$TD_t(s,a) = Q_t^*(s,a) - Q_{t-1}^*(s,a) = \sum_{s' \in S}\left[R\left(s,a,s'\right) + \gamma \max_{a' \in A} Q^*\left(s',a'\right)\right]T\left(s,a,s'\right) - Q_{t-1}^*(s,a) \tag{17}$$

désigne la différence temporelle et $\alpha \in [0\,;\,1]$. Si $\alpha = 0$, l'algorithme n'apprend rien du tout car aucune mise jour ($Q_t^*(s,a) = Q_{t-1}^*(s,a)\ \forall t$) et si $\alpha = 1$, la mise à jour est faite sans tenir compte des valeurs antérieurs. Ainsi $\alpha$ doit appartenir à $]0\,;\,1[$.

## 4.5 Deep Q-learning

### 4.5.1 Principe

En deep Q-learning, on utilise un réseau de neurone pour calculer une approximation de la fonction de qualité $Q$. L'état est donné en entrée du réseau et les Q-valeurs de toutes les actions possibles sont prédictes.[fig.19]

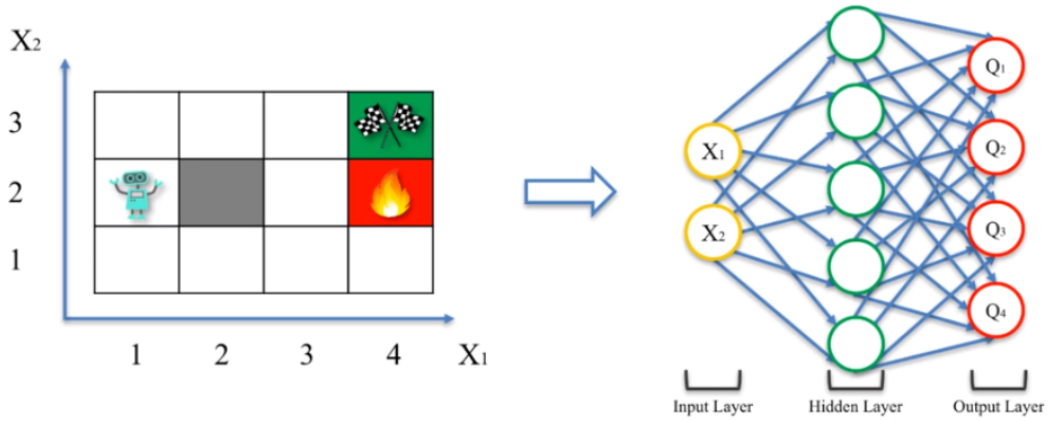



FIGURE 19: Deep Q-learning [19]

Ici, la fonction de coût est l'erreur quadratique entre la valeur prédite $\hat{Q}$ et la valeur cible $Q$. Cependant, les valeurs cibles ne sont pas connues. Les valeurs cibles $Q_t(s,a)$ sont approchées par les valeurs antérieurs $Q_{t-1}(s,a)$. Ainsi à l'itération t, la fonction de coût est donnée par :

$$L = \sum_{a,s} (Q_{t-1}(s,a) - \hat{Q}_t(s,a))^2 \tag{18}$$

où les $\hat{Q}_t(s,a)$ désignent les valeurs prédites par le réseau de neurones.

### 4.5.2 Experience Replay

C'est une technique appliquée pour éviter le sur-apprentissage. En effet, certains états sont rares dans un environnement (par exemple les virages de 90 degré pour une conduite autonome) et cela pose un problème de données déséquilibrées entraînant sur-apprentissage par le réseau de neurones sur

des cas d'états plus fréquents (par exemple aller tout droit pour une conduite autonome). Pour résoudre ce problème, le système enregistre les situations déjà découvertes (états,actions,récompenses,états suivants) dans un tableau et pendant la phase d'apprentissage des données sont choisies aléatoirement dans les données stockées afin de faire revivre le système les situations passées [fig.20]

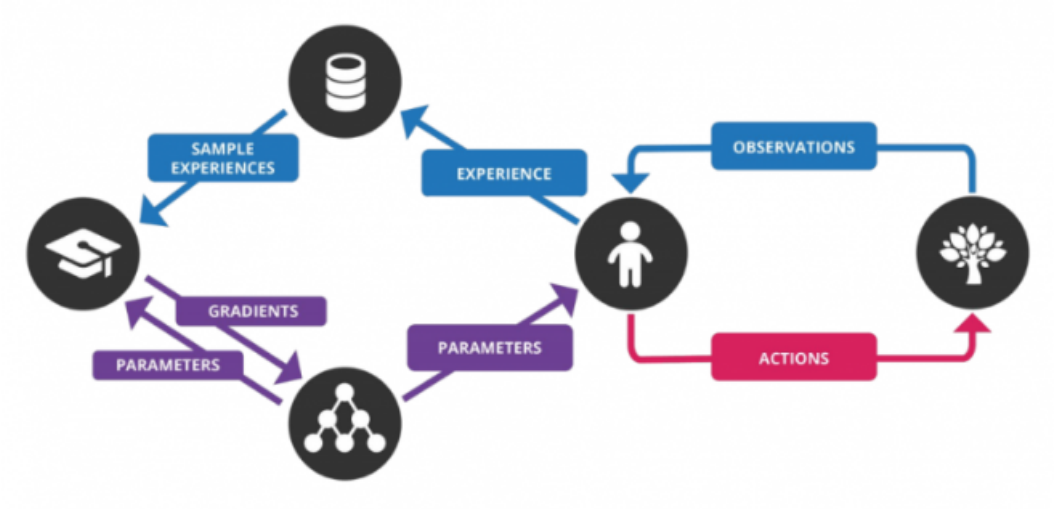



FIGURE 20: Entraînement avec Experience Replay [17]

# A Code source PMC

```
#Importation des librairies utiles
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from imblearn.over_sampling import RandomOverSampler
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
from sklearn.metrics import roc_auc_score,confusion_matrix
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier,AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.gaussian_process import GaussianProcessClassifier

#Importation des données
X=pd.read_csv("dataX.csv")
y=pd.read_csv("datay.csv")
print(X.shape)

# Visualisation de la distribution de la variable cible
y_columns=y.columns
x = y[y_columns[0]].value_counts().values
sns.barplot([0,1], x)
plt.title("Répartition de la variable cible")

# Normalisation des données
scaler = MinMaxScaler(feature_range = (0, 1))
X = scaler.fit_transform(X)

# Séparation des données en données d'apprentissage et de test
X_tr, X_val, y_tr, y_val = train_test_split(X, y, test_size=0.2, random_state=42)
y_val=y_val.iloc[:,:].values
print(len(y_val))

#Augmentations des données
ran=RandomOverSampler()
```

```
X_tr,y_tr= ran.fit_resample(X_tr,y_tr)
print(X_tr.shape)
print(y_tr.shape)
x = np.array([len(y_tr[y_tr==0]),len(y_tr[y_tr==1])])
sns.barplot([0,1], x)
plt.title("Répartition de la variable cible ")

#Construction du dataset d'entrainement du réseau
class Dataset(Dataset):
    def __init__(self, X,y):
        self.X=torch.from_numpy(X).float()
        self.y=torch.tensor(y, dtype=torch.long)
        self.sample=len(y)

    def __len__(self):
        return self.sample

    def __getitem__(self, item):
        return self.X[item],self.y[item]

training_set=Dataset(X_tr,y_tr)
test_set=Dataset(X_val,y_val)
training_generator=DataLoader(training_set, batch_size=32, shuffle=True)
test_generator=DataLoader(test_set, batch_size=32, shuffle=True)

#Construction du PMC
class Network(nn.Module):
    def __init__(self):
        super(Network, self).__init__()
        self.fc1=nn.Linear(in_features=6,out_features=10)
        self.bn1=nn.BatchNorm1d(num_features=10)
        self.fc2=nn.Linear(in_features=10,out_features=10)
        self.bn2=nn.BatchNorm1d(num_features=10)
        self.fc3=nn.Linear(in_features=10,out_features=6)
        self.bn3=nn.BatchNorm1d(num_features=6)
        self.fc4=nn.Linear(in_features=6,out_features=2)
    def forward(self,x):
        x=F.tanh(self.bn1(self.fc1(x)))
        x=F.tanh(self.bn2(self.fc2(x)))
        x=F.tanh(self.bn3(self.fc3(x)))
        x=F.log_softmax((self.fc4(x)),dim=1)

        return x

Net=Network()
print(Net)
```

```
#Entraînement du réseau
criterion = nn.CrossEntropyLoss()
#optimizer = optim.SGD(Net.parameters(), lr=0.001, momentum=0.9,weight_decay= 1e-6)
optimizer = torch.optim.Adam(Net.parameters(), lr=1e-5)
epochs=500
PATH="best_weigths"
test_interval=1
best_loss = 1e10
best_epoch = 0
Net.train()
num_iter_per_epoch = len(training_generator)
test_loss=[]
train_loss=[]
for epoch in range(epochs):
    epoch_loss=[]
    for iter, batch in enumerate(training_generator):
        x, label = batch
        x = Variable(x, requires_grad=True)
        optimizer.zero_grad()
        logits = Net(x)
        loss = criterion(logits, label)
        loss.backward()
        optimizer.step()
        epoch_loss.append(loss*len(label))
    tr_loss=sum(epoch_loss)/len(training_set)
    train_loss.append(tr_loss)
    if epoch % test_interval == 0:
        Net.eval()
        loss_ls = []
        for te_iter, te_batch in enumerate(test_generator):
            te_x, te_label = te_batch
            num_sample = len(te_label)
            with torch.no_grad():
                te_logits = Net(te_x)
            batch_loss= criterion(te_logits, torch.max(te_label, 1)[1])
            loss_ls.append(batch_loss * num_sample)
        te_loss = sum(loss_ls) / test_set.__len__()
        print("Epoch: {}/{} Train_loss:{:.2f} Test_loss:{:.2f}".format(
                epoch + 1,
                epochs,
                tr_loss,
                te_loss
                ))
        test_loss.append(te_loss)
        Net.train()
```

```
        if te_loss < best_loss:
                best_loss = te_loss
                best_epoch = epoch
                torch.save(Net.state_dict(), PATH)

# Affichage des courbes d'erreurs pendant l'entrainement
fig, axs = plt.subplots(1, 2, figsize=(9, 3), sharey=True)
axs[0].plot(np.arange(epochs), train_loss)
axs[0].set_title("Train loss")
axs[1].plot(np.arange(epochs), test_loss)
axs[1].set_title("Test loss")

#Chargement des meilleurs poids
Net.load_state_dict(torch.load(PATH))
Net.eval()

#Evaluation du PMC sur les données de test
x,target=test_set[:]
y_pred=Net(x)
y_pred=torch.argmax(y_pred, dim=1)
cm=confusion_matrix(target,y_pred)
PMC_accuracy=(cm[0][0]+cm[1][1])/sum(sum(cm))
PMC_roc=roc_auc_score(target,y_pred)
print("Acurracy score:",PMC_accuracy)
print("ROC score:",PMC_roc)
print("confusion matrix:")
print(cm)

#Comaparaison du PMC avec d'autres méthodes d'apprentissage statistique

classifiers=[RandomForestClassifier(),GradientBoostingClassifier(),
             AdaBoostClassifier(),DecisionTreeClassifier(),
             KNeighborsClassifier(),GaussianProcessClassifier()]
names=["RandomForestClassifier","GradientBoostingClassifier",
       "AdaBoostClassifier","DecisionTreeClassifier",
       "KNeighborsClassifier","GaussianProcessClassifier"]
Acuracy=[]
ROC=[]

for i in range(len(classifiers)):
    print(names[i]+":")
    classifier=classifiers[i]
    classifier.fit(X_tr,y_tr)
    y_pred=classifier.predict(X_val)
    cm=confusion_matrix(y_val,y_pred)
    accuracy=(cm[0][0]+cm[1][1])/sum(sum(cm))
```

```
    roc=roc_auc_score(y_val,y_pred)
    Acuracy.append(accuracy)
    ROC.append(roc)
    print("Acurracy score:",accuracy)
    print("ROC score:",roc)
    print("confusion matrix:")
    print(cm)
    print("="*20)
names.append("PMC")
Acuracy.append(PMC_accuracy)
ROC.append(PMC_roc)
d={"classifiers":names,"accuracy":Acuracy,"roc score":ROC}
df=pd.DataFrame(data=d)
print(df)
```

# B Code source CNN

```
#Util librairies
!pip install pytorchcv
import torch
from torchvision import transforms, datasets
import matplotlib.pyplot as plt
import numpy as np
from torchvision import models
import torch.nn as nn
import torch.nn.functional as F
import time
import copy
import torch.optim as optim
from torch.optim import lr_scheduler
import pandas as pd
import seaborn as sn
from pytorchcv.model_provider import get_model as ptcv_get_model
import torch
from torch.autograd import Variable
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report


#Load dataset
train_data_transform = transforms.Compose([
        transforms.RandomSizedCrop(224),
        transforms.RandomHorizontalFlip(),
```

```
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225])
    ])
training_set = datasets.ImageFolder(root='../input/covid19xray/covid-19-dataset/train',
                                                transform=train_data_transform)

val_data_transform=transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
test_set = datasets.ImageFolder(root='../input/covid19xray/covid-19-dataset/test',
                                transform=val_data_transform)

dataloaders = {'train':  torch.utils.data.DataLoader(training_set,
                                                batch_size=64, shuffle=True),
               'val':torch.utils.data.DataLoader(test_set,
                                                batch_size=64, shuffle=True)}

dataset_sizes = {'train': len(training_set) ,'val':len(test_set)}
class_names = training_set.classes
print(class_names)


#data exploration
label_0=[0 for i in range(len(training_set)) if training_set[i][1] ==0 ]
label_1=[1 for i in range(len(training_set)) if training_set[i][1] ==1 ]
label_2=[2 for i in range(len(training_set)) if training_set[i][1] ==2]
plt.hist(label_0,histtype='bar',color='red',label=class_names[0])
plt.hist(label_1,histtype='bar',color='green',label=class_names[1])
plt.hist(label_2,histtype='bar',color='orange',label=class_names[2])
plt.legend(prop={'size': 10})
plt.title("Distribution of training set target variable ")

image_1,label_1=training_set[0]
image_2,label_2=training_set[1]
image_3,label_3=training_set[289]
image_4,label_4=training_set[289+1]
image_5,label_5=training_set[289+1095]
image_6,label_6=training_set[289+1095+1]
images=[image_1,image_2,image_3,image_4,image_5,image_6]
labels=[label_1,label_2,label_3,label_4,label_5,label_6]
plt.figure(figsize=(10,6))
for i in range(6):
```

```
    image=images[i]
    image=np.transpose(image,(1,2,0))
    plt.subplot(2,3,i+1)
    plt.imshow(image)
    plt.xticks([])
    plt.yticks([])
    plt.xlabel(class_names[labels[i]])


# Use transfert learning

#Util functions

def train_model(model, criterion, optimizer, scheduler, num_epochs=100):
    since = time.time()
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    losses={'train':[],'val':[]}
    accuracy={'train':[],'val':[]}
    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch+1, num_epochs))
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for iter, batch in enumerate(dataloaders[phase]):
                inputs,labels=batch
                inputs = inputs.to(device)
                labels = labels.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
```

```
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
            if phase == 'train':
                scheduler.step()

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]
            losses[phase]=losses[phase]+[epoch_loss]
            accuracy[phase]=accuracy[phase]+[epoch_acc]

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

        print()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

    # load best model weights
    model.load_state_dict(best_model_wts)
    return {"model":model,'losses':losses,'accuracy':accuracy}


def tensorbord(epoch,results):
    epochs=np.arange(1,epoch+1)
    losses=results["losses"]
    accuracy=results["accuracy"]
    plt.figure(figsize=(15,5))
    plt.subplot(1,2,1)
    plt.plot(epochs,losses["train"],label="train",color="orange")
```

```
    plt.plot(epochs,losses["val"],label="val",color="green")
    plt.xlabel("epoch")
    plt.ylabel("loss")
    plt.title("Train/Val losses")
    plt.legend()
    plt.subplot(1,2,2)
    plt.plot(epochs,accuracy["train"],label="train",color="orange")
    plt.plot(epochs,accuracy["val"],label="val",color="green")
    plt.xlabel("epoch")
    plt.ylabel("accuracy")
    plt.title("Train/Val accuracy")
    plt.legend()
    return

def visualize_prediction(model, num_images=9):
    was_training = model.training
    model.eval()
    images_so_far = 0
    fig = plt.figure(figsize=(12,11))

    with torch.no_grad():
        for i, (inputs, labels) in enumerate(dataloaders['val']):
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            scores = F.softmax(outputs,dim=1)
            cfs, _ = torch.max(scores, 1)

            for j in range(inputs.size()[0]):
                images_so_far += 1
                plt.subplot(3,3,images_so_far)
                image=inputs.cpu().data[j]
                image=np.transpose(image,(1,2,0))
                plt.imshow(image)
                plt.xticks([])
                plt.yticks([])
                plt.xlabel("actual: "+str(class_names[labels[j]]),color="green")
                plt.title('predicted: {} {}%'.format(class_names[preds[j]],round(float(
                          color="orange")
                if images_so_far == num_images:
                    model.train(mode=was_training)
                    return
        model.train(mode=was_training)
```

```
def evaluate(model):
    model.eval()
    data=torch.utils.data.DataLoader(test_set,batch_size=len(test_set), shuffle=True)
    with torch.no_grad():
            for iter,batch in enumerate(data):
                images,labels=batch
                images=images.to(device)
                labels=labels.to(device)
                outputs = model(images)
                _, preds = torch.max(outputs, 1)
                print(classification_report(labels.cpu(),  preds.cpu(), target_names=cl
                cm=confusion_matrix(labels.cpu(),preds.cpu())
                df_cm = pd.DataFrame(cm, index = class_names,columns = class_names)
                plt.figure(figsize = (10,5))
                sn.heatmap(df_cm, annot=True)

    return


# First model resnet18
resnet = models.resnet18(pretrained=True)
num_ftrs = resnet.fc.in_features
resnet.fc = nn.Linear(num_ftrs, 3)
resnet=resnet.cuda()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(resnet.parameters(), lr=0.001, weight_decay=0.0005)
#optimizer = optim.SGD(resnet.parameters(), lr=0.001, momentum=0.9)
# Decay LR by a factor of 0.1 every 10 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
results = train_model(resnet, criterion, optimizer, exp_lr_scheduler,
                      num_epochs=100)
resnet=results["model"]
torch.save(resnet, "resnet-covid-19-model.pth")
torch.save(resnet.state_dict(),"resnet-covid-19-weigths.pth" )

tensorbord(100,results)

visualize_prediction(model=resnet, num_images=9)

evaluate(resnet)


#2nd model darknet53
darknet53 = ptcv_get_model("darknet53", pretrained=True)
darknet53.output=nn.Linear(1024, 3)
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()
darknet53=darknet53.to(device)
optimizer = torch.optim.Adam(darknet53.parameters(), lr=0.001, weight_decay=0.0005)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)

results_darknet53 = train_model(darknet53, criterion, optimizer, exp_lr_scheduler,
                        num_epochs=100)
darknet53=results_darknet53["model"]
torch.save(darknet53, "darknet53-covid-19-model.pth")
torch.save(darknet53.state_dict(),"darknet53-covid-19-weigths.pth" )

tensorbord(100,results_darknet53)

visualize_prediction(model=darknet53, num_images=9)

evaluate(darknet53)
```

## C Code source cartes de Kohonen

```
# Importation des librairies nécessaires
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.metrics import confusion_matrix
import missingno as msno
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import MinMaxScaler
from minisom import MiniSom
from pylab import bone, pcolor, colorbar, plot, show

#Importation des données
data=pd.read_csv("word_countries.csv")
print(data.shape)
data.head()

#Attribution d'identifiants aux différents pays du dataset
ID=[i for i in range(155015,155015+len(data))]
coresp={k:v for k,v in zip(ID,data["Country"])}
Country=list(coresp.values())
data.insert(2, "ID", ID)
data=data.drop(labels=["Country","Region"],axis=1)
data.head()
```

```
print(data.dtypes)

#Convertion des valeurs nulériques string en float
def convert(x):

    return float(x.replace(',','.'))

print(convert("6,5"))
columns=data.loc[:, data.dtypes == object]
columns=columns.columns
columns=list(columns)
data[columns] = data[columns].astype(str)
for a in columns:
    data[a]=data[a].apply(lambda x: convert(x))
print(data.dtypes)
data.dtypes

#Affichage des valeurs manquantes de chaque colonne
msno.bar(data)

#Traitement des valeurs manquantes
imputer = SimpleImputer(missing_values=np.nan, strategy="mean")
X=imputer.fit_transform(data)
print(X.shape)

#Normalisation des données
scaler = MinMaxScaler(feature_range = (0, 1))
X = scaler.fit_transform(X)

#Construction et entraînement de la SOM
som = MiniSom(x = 5, y = 4, input_len = 19, sigma = 0.8, learning_rate = 0.5)
som.random_weights_init(X)
som.train_random(data = X, num_iteration = 1000)

#Affichage de la carte de sortie
bone()
pcolor(som.distance_map().T)
colorbar()

#Pays derrières chaque neurone de sortie
mapps = som.win_map(X)
keys=list(mapps.keys())
values=list(mapps.values())
values=[len(v) for v in values ]
result={key:value for key,value in zip(keys,values)}
print(result)
```

```
bone()
pcolor(som.distance_map().T)
colorbar()
dictio={}
j=0
for k,v in mapps.items():
    j=j+1
    plt.text(k[0]+0.5,k[1]+0.5,"G"+str(j),color="red")
    liste=[]
    for i in range(len(v)):
        country=v[i]
        country=country.reshape(1, -1)
        country=scaler.inverse_transform(country)
        country=coresp[country[0][0]]
        liste.append(country[:4])
    dictio["Groupe"+str(j)]=liste
print(dictio)
df=pd.DataFrame.from_dict(dictio, orient='index')
df.head()
```

# 5 Références


[1] Y. Li, *REINFORCEMENT LEARNING APPLICATIONS*, arXiv:1908.06973

[2] M. van Otterlo and M. Wiering, *Reinforcement Learning and Markov Decision Processes*

[3] T. Schaul, J. Quan, I. Antonoglou and D. Silver, *PRIORITIZED EXPERIENCE REPLAY*

[4] M. Carrasco Kind and Robert J. Brunner, *SOMz : photometric redshift PDFs with self organizing maps and random atlas*

[5] , https://www.kaggle.com/fernandol/countries-of-the-world

[6] , B. Krose, P. van der Smagt, *An introduction to Neural Networks*

[7] http://odds.cs.stonybrook.edu/annthyroid-dataset/

[8] https://pytorch.org/

[9] Andrew P.Bradley, *The use of the area under the ROC curve in the evaluation of machine learning algorithms*

[10] https://test.pypi.org/project/MiniSom/1.0/

[11] https://docs.gimp.org/2.6/fr/plug-in-convmatrix.html

[12] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola,*Dive into Deep Learning,Release 0.7.0*

[13] https://www.mathworks.com

[14] Cours de spécialisation deep learning d'Andrew Ng sur Coursera, https://www.coursera.org/specializations/deep-learning

[15] M.E.H. Chowdhury, T. Rahman, A. Khandakar, R. Mazhar, M.A. Kadir, Z.B. Mahbub, K.R. Islam, M.S. Khan, A. Iqbal, N. Al-Emadi, M.B.I. Reaz, "Can AI help in screening Viral and COVID-19 pneumonia ?" arXiv preprint, 29 March 2020, https://arxiv.org/abs/2003.13145. https://www.kaggle.com/tawsifurrahman/covid19-radiography-database

[16] Wouter M. Kouw, M. Loog , *An introduction to domain adaptation and transfer learning*

[17] https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

[18] https://www.superdatascience.com/blogs/the-ultimate-guide-to-self-organizing-maps-so

[19] https://www.udemy.com/course/intelligence-artificielle-az/learn/lecture/10858968?sta

[20] https://en.wikipedia.org/wiki/Reinforcement$_l earning/media/File : Reinforcement_l earning_d iagram.su