

M2 INGÉNIERIE STATISTIQUE ET DATA SCIENCE

RÉSEAUX DE NEURONES ARTIFICIELS

## Rapport de projet

RAPHAËL MIGNOT<sup>1</sup>



Sorbonne Université  
Institut de Statistique de l'Université de Paris

PROFESSEUR : ANNICK VALIBOUZE<sup>2</sup>

Mars 2020

---

<sup>1</sup>mail : [raphael.mignot@etu.upmc.fr](mailto:raphael.mignot@etu.upmc.fr), LinkedIn : [raphael-mignot](https://www.linkedin.com/in/raphael-mignot)

<sup>2</sup>Membre du LIP6 et du LPSM. Page personnelle : <https://www-apr.lip6.fr/~avb/>

# Contents

<b>1 Réseaux neuronaux et le cas du perceptron multi-couches</b>	<b>3</b>
1.1 Architecture . . . . .	3
1.2 Apprentissage du perceptron multi-couches . . . . .	5
1.2.1 Rétro-propagation . . . . .	6
1.2.2 Mise à jour des poids et des biais . . . . .	7
1.2.3 Conclusion sur l'apprentissage . . . . .	8
1.3 Sur-apprentissage et régularisation en apprentissage profond . . . . .	8
1.3.1 Pénalisation des hyperparamètres . . . . .	9
1.3.2 Arrêt prématuré . . . . .	9
1.3.3 Autres méthodes de régularisation . . . . .	10
1.4 Conclusion . . . . .	11
<b>2 Perceptron multi-couches sous R</b>	<b>11</b>
2.1 Présentation du jeu de données MNIST . . . . .	11
2.2 Utilisation du package <code>nnet</code> . . . . .	12
2.3 Comparaison avec une autre méthode statistique : la forêt aléatoire . . . . .	13
<b>3 Perceptron multi-couches avec le logiciel Tensorflow</b>	<b>14</b>
3.1 Construction de l'architecture . . . . .	15
3.2 Phase d'apprentissage . . . . .	15
<b>4 Carte de Kohonen (ou carte auto-adaptative)</b>	<b>16</b>
4.1 Principe . . . . .	16
4.2 Utilisation pratique . . . . .	18
<b>5 Machine de Boltzmann restreinte</b>	<b>19</b>
5.1 Présentation . . . . .	19
5.2 Apprentissage et divergence contrastive . . . . .	21
5.3 Application à la classification . . . . .	23
<b>6 Réseaux de neurones convolutionnels</b>	<b>24</b>
6.1 Principe . . . . .	24
6.1.1 Convolution . . . . .	24
6.1.2 Groupage (ou pooling) . . . . .	25
6.2 Utilisation pratique d'un réseau convolutionnel . . . . .	26
<b>7 Conclusion du rapport</b>	<b>29</b>
<b>A Code des script R et Python</b>	<b>30</b>
A.1 Perceptron multi-couches sous R et comparaison avec l'algorithme forêt aléatoire	30
A.2 Perceptron multi-couche (PMC), carte de Kohonen (SOM), machine de Boltzmann restreinte (RBM) et réseau convolutif (CNN) sous Python . . . . .	32

## Abstract

Ce rapport a été rédigé dans le cadre du cours de "Réseaux de neurones" du Master 2 *Ingénierie Statistique et Data Science* de l'Institut de Statistique de l'Université de Paris (ISUP)<sup>3</sup>, école rattachée au département de sciences de Sorbonne Université<sup>4</sup>. Ce cours a été enseigné par Annick Valibouze<sup>5</sup> de janvier à mars 2020.

Nous y présentons quelques méthodes d'apprentissage profond. Nous abordons les points de vue théorique de ces algorithmes ainsi que pratique. Pour la partie théorique, nous avons tâché d'être le plus clair possible tout en restant exact mathématiquement et assez concis. La partie pratique s'est effectuée majoritairement avec le logiciel TensorFlow sous Python 3 (une section présente l'utilisation du perceptron sous R) dans le cadre de la reconnaissance d'image (vision artificielle). Les blocs de code importants sont présenté au fur et à mesure du rapport. On trouvera en annexe les codes dans leur entiereté, sous la forme notebook.

## Mots clés.

Apprentissage profond, Apprentissage supervisé, Apprentissage non-supervisé, Perceptron multi-couches, Méthode de la rétro-propagation, Estimateur universel, Régularisation, Carte de Kohonen (Carte auto-adaptative, *Self-organizing maps*), Machine de Boltzmann restreinte, Divergence Contrastive, Réseaux convolutionnels, classification, vision artificielle, TensorFlow.

---

<sup>3</sup>Lien vers le site de l'Institut de Statistique de l'Université de Paris

<sup>4</sup>Lien vers le site de Sorbonne Université

<sup>5</sup><https://www-apr.lip6.fr/~avb/>

# 1 Réseaux neuronaux et le cas du perceptron multi-couches

Le perceptron multi-couches est le modèle de base dans le domaine de l'apprentissage profond et des réseaux de neurones artificiels. Le but est d'estimer la relation entre des données d'entrée  $\mathbf{x}$  et des données de sortie correspondantes  $\mathbf{y}$ . On modélise cette relation par une fonction  $f^*$ . Nous avons  $\mathbf{y} = f^*(\mathbf{x})$  et notre but est de trouver une fonction  $f$  le plus proche possible de  $f^*$ , au sens de la fonction de perte  $l$  choisie, à partir de la donnée de  $(\mathbf{x}, \mathbf{y})$ . On veut minimiser en  $f$  :

$$l(f(\mathbf{x}), \mathbf{y})$$

## 1.1 Architecture

Le perceptron multi-couches est constitué d'une succession de *couches* de largeurs fixées. Chaque couche est composée d'*unités* (aussi appelées neurones) et sa largeur correspond au nombre d'unités qu'elle contient. Il existe trois types de couches. La couche d'entrée, la couche de sortie et le reste, que l'on appellera couches *cachées* (elles se situent entre les deux précédentes). Des arêtes orientées (vers la couche de sortie) relient les couches.

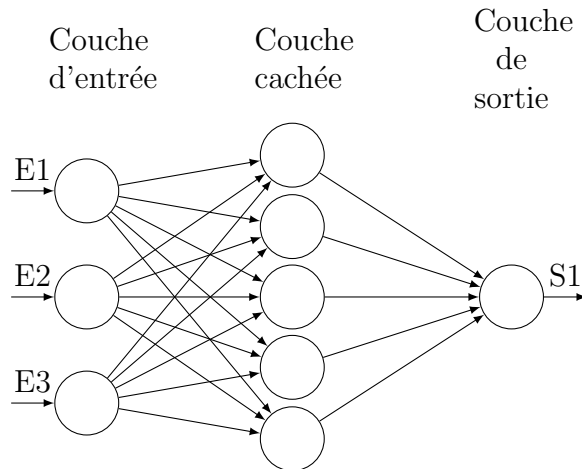


Figure 1: Architecture d'un perceptron à une couche cachée.

Chaque unité est associée à deux opérations : une **transformation linéaire** de l'entrée et l'application d'une fonction dite d'**activation**. Cette dernière est une fonction mesurable que l'on sait dériver. Généralement, cette fonction est continue et d'une forme analytique très simple. Par exemple une fonction très utilisée est  $\text{ReLU}(x) = \max\{0, x\}$  qui se trouve être continue et dérivable presque partout sur  $\mathbb{R}^d$ .

Lorsqu'une entrée  $x$  accède à une unité du réseau, elle subit une transformation linéaire  $z = W^T x + b$ . On notera  $w_{i,j}$  les coefficients de la matrice  $W$ . Puis, on applique la fonction

d'activation  $y = g(z)$  et on obtient une sortie  $y$ .

**Remarque.** On utilise généralement la même fonction d'activation au sein d'une couche. Cependant, rien ne nous interdirait d'avoir une fonction d'activation différente par unité.

Voici quelques exemples de fonctions d'activations souvent utilisées :

- ReLU (acronyme de *Rectified Linear Unit*) :  $g(z) = \max(0, z)$
- ReLU généralisé :  $g(z; \alpha) = \max(0, z) + \alpha \min(0, z)$
- Correction valeur absolue :  $g(z) = |z|$
- *Leaky* ReLU : ReLU généralisé avec  $\alpha$  petit (de l'ordre de  $10^{-2}$ )
- PReLU (*Parametric* ReLU) :  $\alpha$  est un paramètre à apprendre
- Sigmoid / tangente hyperbolique :  $g(z) = \sigma(z) := \frac{1}{1+e^{-z}}$  et  $g(z) = \tanh(z)$ . Les deux sont très liées puisque  $\tanh(z) = 2\sigma(2z) - 1$ . Les réseaux qui contiennent l'une ou l'autre ont des propriétés assez similaires.
- Unités maxout
- Linéaire :  $g(z) = z$ . À noter que si toutes les fonctions d'activations sont linéaires le réseau produit alors un estimateur linéaire. On rencontre parfois cette fonction d'activation sur certaines couches. Cela revient à diviser une simple couche en deux couches : la première étant celle avec  $g_1(z) = z$ . Soit  $x$  le vecteur d'entrée. Après passage dans la couche 1 on a  $U^T x + b_1$ . Puis on passe dans la couche 2 :  $h = g_2(V^T U^T x + b_1 + b_2)$  ie  $h = g(W^T x + b)$  avec  $g := g_2$ ,  $W^T := V^T U^T$  et  $b := b_1 + b_2$ . Si on note les dimensions comme suit :  $U \in \mathbb{R}^{n \times q}$  et  $V \in \mathbb{R}^{q \times p}$ , on a dans le cas à une couche  $np$  paramètres à déterminer. Dans le cas à deux couches, on a  $(n + p)q$  paramètres à estimer. Lorsque  $q$  est petit, l'utilisation de couches linéaires réduit le nombre de paramètres du réseau dans son ensemble.
- Fonction radiale de base (RBF) :  $h_i = \exp\left(-\frac{1}{\sigma_i^2} \|W_{ii} - x\|^2\right)$ . Cette fonction d'activation est particulière, elle est utilisée dans les réseaux dits RBF et les neurones qui l'utilisent n'effectuent pas de transformation linéaire au préalable.
- Softplus :  $g(z) = \xi(z) := \log 1 + e^z$
- Hard tanh :  $g(z) = \max(-1, \min(1, z))$
- Softmax : cette fonction d'activation est utilisée sur la couche de sortie des réseaux qui effectuent de la classification. Soit  $K$  le nombre de classes. On a nos  $K$  activations  $\mathbf{a} = (a_1, \dots, a_K)^T$ .

$$h_i = \text{softmax}(a_i) := \frac{e^{a_i}}{\sum_{j=1}^K e^{a_j}}$$

Ainsi, à partir de valeurs réelles, nous obtenons des valeurs comprises entre 0 et 1 dont la somme vaut 1.  $\forall i = 1, \dots, K, h_i$  est interprété comme la probabilité que l'entrée

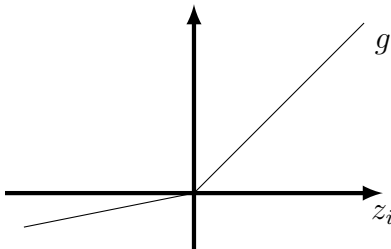


Figure 2: Un exemple de fonction d'activation très utilisé : le *leaky* ReLU ( $\alpha = 0.2$ ).

considérée appartenir à la classe  $i$ . Tous les réseaux actuels de classification utilisent cette fonction d'activation sur leur couche de sortie.

C'est cet ensemble d'opérations que l'on appelle réseau de neurones. Bien qu'étant d'apparence simple, cet algorithme est en réalité un **estimateur universel** : toute fonction mesurable  $f$  peut être approximée d'aussi près que l'on veut par un perceptron à une couche cachée (avec la plupart des fonctions d'activations, par exemple le sigmoïde) pourvu que l'on puisse élargir d'autant que nécessaire la couche cachée et que l'espace de départ de  $f$  soit de dimension finie<sup>6</sup>.

L'architecture d'un réseau dépend de son nombre de neurones et de la façon dont on les connecte entre eux. L'architecture la plus simple est celle par chaîne (comme sur la figure 1 : c'est le perceptron multi-couches). Par ailleurs, on pourrait ne pas connecter tous les neurones d'une couche  $i$  à ceux de la couche  $i + 1$ . On rencontre aussi des réseaux dans lesquels des unités de la couche  $i$  sont directement connectées à des unités de la couche  $i + 2$  (on saute une couche).

**Remarque.** Chaque domaine d'application utilise des architectures spécifiques. Par exemple, en reconnaissance d'image une architecture répandue est le réseau de neurones convolutif que nous présentons dans la section 6.

## 1.2 Apprentissage du perceptron multi-couches

On note  $J$  la fonction de coût. Le but est de minimiser cette fonction de sorte à être le plus près possible des antécédents dont nous disposons (les  $y_i$ ) au sens de la distance choisie. Par exemple, si nous effectuons de la reconnaissance d'image sur une photo de chien, nous souhaitons sortir une probabilité la plus proche de 1 possible pour la catégorie chien, et proche de 0 pour les autres catégories. On va pour ce faire utiliser les méthodes classiques d'optimisation comme les méthodes de descentes. Ces méthodes sont très performantes mais nécessitent le calcul du gradient :  $\nabla_{\theta} J$  où  $\theta$  est le vecteur regroupant tous les paramètres (ie les poids  $w_{ij}$  et les biais  $b_i$ ).

---

<sup>6</sup>[Hornik et al., 1989], [Cybenko, 1989]

On dit que l'on *propage*  $x$  à travers le réseau lorsqu'on obtient la sortie associée  $\hat{y}$  à l'aide de laquelle on calcule le coût  $J(\theta)$ . Par exemple

$$J(\theta) = \frac{1}{2n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

l'erreur des moindres carrés, mais cela pourrait être

$$J(\theta) = -\frac{1}{n} \sum_{i=1}^n [y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i)]$$

l'entropie croisée dans le cas d'une classification binaire (avec  $y_i = 0$  ou  $1$  et  $\hat{p}_i$  la probabilité estimée d'être dans la classe  $1$ ).

### 1.2.1 Rétro-propagation

La rétro-propagation est une méthode pour obtenir ce gradient qui a l'avantage d'être peu coûteuse en calculs. Elle se base sur le théorème de dérivation des fonctions composées. On se place dans le cas d'un PMC complètement connecté (chaque neurone d'une couche est connecté à chaque neurone de la couche voisine). On parle parfois de réseau dense. Notons  $w_{ij}^k$  le poids entre le neurone  $j$  de la couche  $k$  et le neurone  $i$  de la couche  $k - 1$ . Nous cherchons à différentier  $J$  selon  $\theta$  le vecteur des paramètres, c'est-à-dire selon chacun des  $w_{i,j}^k$ . Nous appliquons le théorème sur cette dérivée :

$$\frac{\partial J}{\partial w_{i,j}^k} = \frac{\partial J}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{i,j}^k}$$

où  $a_j^k$  est l'activation du neurone  $j$  de la couche  $k$  :  $a_j^k = W^T x + b$ . On note  $h_i^k$  la sortie du neurone  $i$  de la couche  $k$ , donc  $h_i^k = g_k(a_i^k)$  avec  $g_k$  la fonction d'activation de la couche  $k$ . On a

$$a_i^k = b_i^k + \sum_j w_{ji}^k h_j^{k-1}$$

La notation est assez lourde, l'idée assez simple : le neurone  $i$  de la couche  $k$  reçoit en entrée toutes les sorties des neurones de la couche  $k - 1$ . Ces sorties forment un vecteur dont on fait le produit scalaire avec la  $i^e$  ligne de  $W^T$  et on ajoute le biais pour obtenir l'activation du neurone. Pour simplifier, on intègre le biais dans la matrice  $W$ . A partir de cette équation on obtient  $\forall i, j, k$ ,

$$\frac{\partial a_j^k}{\partial w_{i,j}^k} = \frac{\partial}{\partial w_{i,j}^k} \sum_l w_{lj}^k h_l^{k-1} = h_i^{k-1}$$

Donc on a

$$\frac{\partial J}{\partial w_{i,j}^k} = h_i^{k-1} \frac{\partial J}{\partial a_j^k}$$

Nous supposons que nous avons gardé en mémoire les valeurs des  $h_i^k, \forall i, k$ , calculés lors de la propagation. Il ne s'agit plus que de déterminer  $\frac{\partial J}{\partial a_j^k}, \forall j, k$ . Pour cela, nous allons procéder

couche par couche, en commençant par la dernière, puis l'avant-dernière, etc. Notons  $K$  le nombre de couches, on calcule donc tout d'abord  $\frac{\partial J}{\partial a_j^K}$ .

Pour le cas particulier de l'erreur des moindres carrés, on obtiendrait

$$\frac{\partial J}{\partial a_j^K} = (y - \hat{y})g'_K(a_j^K)$$

pour la couche de sortie du réseau. Puis pour les couches cachées, on applique de nouveau le théorème de dérivation des fonction composées :

$$\boxed{\frac{\partial J}{\partial a_j^k} = \sum_l \frac{\partial J}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial a_j^k}}$$

En d'autres mots : la dérivée par rapport à une activation précise d'une couche  $k$  peut être déterminée à partir des dérivées par rapport aux activations de la couche suivante  $k + 1$  et des dérivées de ces activations par rapport à notre activation de la couche  $k$ . C'est en réalité une petite astuce qui permet de faire une **différentiation très rapide** car beaucoup de calculs sont stockés et ré-utilisés plusieurs fois. Pour ce qui est du terme tout à droite, nos cours de dérivations de l'enseignement secondaire nous disent que :

$$\frac{\partial a_l^{k+1}}{\partial a_j^k} = w_{jl}^{k+1} g'(a_j^k)$$

puisqu'on rappelle que  $a_l^{k+1} = \sum_j w_{jl}^{k+1} g(a_j^k)$ .

**Remarque.** Il est donc d'autant plus intéressant d'utiliser des fonctions d'activations dont les dérivées sont très simples à calculer. Par exemple pour ReLU on a

$$g'(x) = \mathbb{1}_{\mathbb{R}^+}(x)$$

**Remarque.** La rétro-propagation est une méthode de différentiation parmi beaucoup d'autres. C'est celle qui est utilisée dans le domaine des réseaux neuronaux.

### 1.2.2 Mise à jour des poids et des biais

A ce stade, on a la donnée de  $\nabla_{\theta} J$ . On souhaite modifier la matrice des poids  $W$  et les biais  $b$  de manière à diminuer le coût total  $J$ . Toute méthode d'optimisation utilisant le gradient est envisageable. Par exemple, la descente de gradient (aussi appelée méthode de la plus profonde descente) :

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial J(x, \theta^t)}{\partial \theta}$$

avec  $\alpha$  le pas.



### 1.2.3 Conclusion sur l'apprentissage

Voici les étapes de l'apprentissage d'un perceptron multi-couches :

1. Effectuer la propagation et obtenir le coût  $J(x, \theta)$ .
2. Effectuer la rétro-propagation et obtenir le gradient  $\nabla_{\theta} J$ .
3. Mettre à jour les poids et les biais selon la méthode d'optimisation choisie.

Ces trois étapes correspondent à ce que l'on appelle une époque.

## 1.3 Sur-apprentissage et régularisation en apprentissage profond

Il est courant que le réseau de neurones s'adapte de trop près à nos données et qu'il se généralise mal. Cela se traduit par une erreur d'entraînement très faible et une erreur de test élevée comme c'est le cas dans la zone de sur-apprentissage de la figure 3. Toute méthode qui réduit l'erreur de test sans réduire celle d'entraînement est appelée **méthode de régularisation** (d'ailleurs, elles font en général augmenter l'erreur d'entraînement).

Nous nous attardons quelque peu sur ce sujet de la régularisation car bien souvent ces méthodes améliorent grandement la généralisation de nos algorithmes à de nouvelles données (en apprentissage automatique de manière générale, pas uniquement dans le contexte des réseaux de neurones). Ces procédures ont l'avantage de pouvoir s'appliquer à tout type d'algorithme d'apprentissage automatique. Par ailleurs, il arrive que certaines de ces méthodes ne soient volontairement pas utilisés dans un article scientifique pour ainsi mieux pouvoir comparer l'algorithme en question avec ce qui se fait déjà dans le domaine.

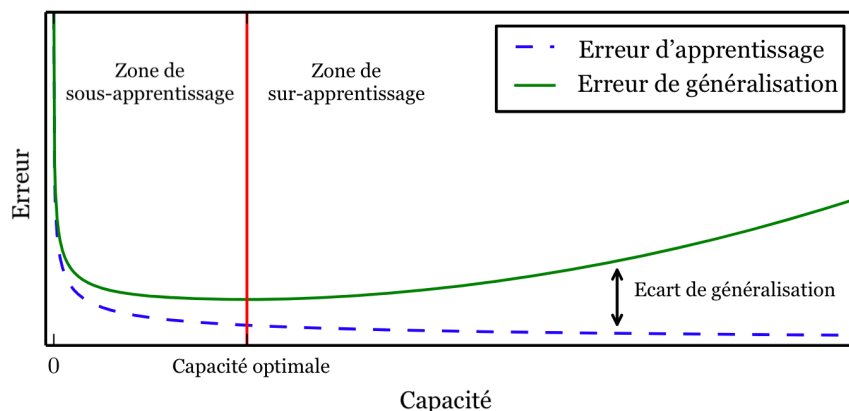


Figure 3: Relation entre erreur d'entraînement et erreur de test. La régularisation vise à passer d'une capacité en sur-apprentissage à une capacité optimale. Traduction d'un schéma de [Goodfellow et al., 2016].

### 1.3.1 Pénalisation des hyperparamètres

Une méthode basique de régularisation est d'introduire une pénalisation sur l'hyperparamètre à apprendre  $\theta$ . Formellement, on crée un coût total  $\tilde{J}$  :

$$\tilde{J}(\theta, \mathbf{x}, \mathbf{y}) = J(\theta, \mathbf{x}, \mathbf{y}) + \alpha \Omega(\theta)$$

avec  $\alpha > 0$  fixé,  $\Omega$  la fonction de pénalité.

**Remarque.** En apprentissage profond, ce sont les poids  $w_i$  qui sont pénalisés et rarement les biais  $b_i$ .

Voici des exemples de pénalités :

- Régression d'arête (*Ridge Regression* ou encore *weight decay*) :  $\Omega(\theta) = \frac{1}{2} \|\mathbf{w}\|_2^2$ . Ainsi les poids sont rapprochés de l'origine de l'espace.
- Régularisation  $L^1$  :  $\Omega(\theta) = \|\mathbf{w}\|_1$ . Pour le modèle linéaire et le coût des moindres carrés cela correspond au Lasso<sup>7</sup>. Même chose que précédemment, les poids sont rapprochés de l'origine. Par ailleurs, cela mène à une sélection de variables puisque certains poids seront réglés sur 0. Il est observé que cette régularisation donne une solution  $\theta$  plus parcimonieuse qu'avec la régression d'arête.

**Remarque.** Plus  $\alpha$  est grand, plus le modèle sera simplifié (avec des  $w_i$  rapprochés de l'origine de l'espace).

La plupart des logiciels permettent d'effectuer une pénalisation aisément. Par exemple, avec Tensorflow et la surcouche Keras que nous allons utiliser dans la section 3, cela s'effectue couche par couche de la manière suivante :

RÉGULARISATION AVEC KERAS : PÉNALISATION PAR RÉGRESSION D'ARÊTE.

---

```
from keras.regularizers import l2
...
model.add(Dense(32, kernel_regularizer=l2(0.01)))
```

---

L'utilisation de Keras sera détaillée dans la section 3.

### 1.3.2 Arrêt prématuré

Une autre méthode couramment utilisée est l'arrêt prématuré (*Early Stopping*). Cela consiste à enregistrer les hyperparamètres à chaque époque et évaluer à intervalle d'époque régulier l'erreur de généralisation. Dès lors qu'on obtient une augmentation de l'erreur de généralisation, l'algorithme est arrêté prématurément.

Voici un petit bout de code qui permet d'utiliser la méthode de l'arrêt prématuré avec Keras.

RÉGULARISATION AVEC KERAS : L'ARRÊT PRÉMATURÉ.

---

<sup>7</sup>[Tibshirani, 1996]

---

```

from keras.callbacks.callbacks import EarlyStopping
callback = EarlyStopping(monitor='val_loss', patience=5)
# This callback will stop the training when there is no
# improvement in the validation loss for five
# consecutive epochs.
model.fit(data, labels, epochs=100, callbacks=[callback],
          validation_data=(val_data, val_labels))

```

---

Ces méthodes sont très simples à ajouter à son modèle et bien souvent elles permettent de diminuer grandement l'erreur de généralisation.

### 1.3.3 Autres méthodes de régularisation

Les deux méthodes précédentes sont constamment utilisés dans le cadre des réseaux de neurones. Il existe de nombreuses autres méthodes de régularisation. On peut notamment citer<sup>8</sup> :

- **Représentation parcimonieuse** de l'activation  $\mathbf{h}$ . Il s'agit d'ajouter une pénalité non pas sur l'hyperparamètre  $\theta$  mais sur l'activation :  $\tilde{J}(\theta, \mathbf{x}, \mathbf{y}) = J(\theta, \mathbf{x}, \mathbf{y}) + \alpha \Omega(\mathbf{h})$  avec  $\alpha > 0$  et par exemple  $\Omega(\mathbf{h}) = \|\mathbf{h}\|_1$  norme qui a pour effet, comme avec l'hyperparamètre, de rendre  $\mathbf{h}$  parcimonieux (*sparse*).
- **Bagging** et méthodes ensemblistes de manière générale. Dans le cas du *bagging* (pour *Bootstrap Aggregating*), il s'agit d'effectuer une "combinaison" d'estimateurs dit faibles (*weak learners*). Un estimateur faible est un estimateur qui a de meilleur résultat qu'un lancer de pièce en classification, ou qu'une moyenne sur toutes les sorties en régression. Ces estimateurs sont entraînés sur des sous-échantillons différents du jeu de données (créés par tirages avec remise). Ainsi, ils auront des caractéristiques différentes et n'effectueront pas forcément les mêmes erreurs. Une combinaison prends la forme d'une moyenne de ces estimateurs en régression, et d'un choix par vote majoritaire en classification.
- **Dropout**, qui consiste à supprimer aléatoirement des unités du réseau (généralement en multipliant la sortie de l'unité par 0).
- **Partage d'hyperparamètres**. Cela consiste à ajouter une pénalité de la forme  $\Omega(\theta) = \|\mathbf{w} - \tilde{\mathbf{w}}\|_2^2$  où  $\tilde{\mathbf{w}}$  est un hyperparamètre que l'on aurait trouvé lors d'une autre procédure et qu'on pense être proche de la solution  $\mathbf{w}^*$ . Dans le même ordre d'idée, on peut effectuer une pénalisation en fixant certains hyperparamètres égaux à d'autres hyperparamètres (au lieu de chercher à ne pas s'en éloigner comme avec une pénalité  $L^2$ ). Cette méthode est constamment utilisée dans le domaine de la vision artificielle.

---

<sup>8</sup>Pour plus de détails, le lecteur peut se référer à [Goodfellow et al., 2016], chapitre 7 : *Regularization for Deep Learning*.

## 1.4 Conclusion

Pour résumer, voici les principales caractéristiques d'un réseau de neurones :

- Son architecture (PMC, récurrent, GAN, etc.). Combien de neurones et comment sont-ils connectés ? Quelle fonction d'activation pour chaque couche ?
- Sa fonction de coût, aussi appelée fonction de perte ou d'erreur (moindres carrés, Hinge, entropie croisée, etc.). Parfois, on parle de coût total : cela correspond au coût et l'ajout de la pénalité  $\Omega(\theta)$ .
- Sa méthode d'optimisation (gradient stochastique, ADAM, etc.).
- Sa métrique (justesse, ROC AUC, F-score, etc.) pour évaluer les performances du réseau et le comparer à d'autres algorithmes.

**Remarque.** Pour toute méthode d'apprentissage automatique, les performances de celle-ci dépendent de la métrique utilisée. Prenons l'exemple d'une classification binaire sur des données non équilibrées : pensons à une détection de fraudes à la carte bancaire, il y a bien plus de non-fraudes que de fraudes. Ainsi, pour la métrique justesse

$$\text{Justesse} = \frac{\text{Nb de prédictions correctes}}{\text{Nb d'observations}}$$

un classifieur qui consisterait à mettre toutes les observations dans la case non-fraude aurait un score très élevé et sûrement meilleur qu'un réseau de neurones aussi complexe soit-il. Cependant, il ne résoudrait absolument pas le problème et il faut donc choisir une métrique adaptée.

## 2 Perceptron multi-couches sous R

Il existe plusieurs solutions pour créer des réseaux de neurones sous R. On peut notamment citer les packages `nnet`, `h2o`, `DeepNet` et `kerasR`. Nous utiliserons dans cette partie le package `nnet`.

**Remarque.** Tous les scripts (R, Python) utilisés sont disponibles **dans leur entièreté** en annexe.

### 2.1 Présentation du jeu de données MNIST

MNIST est un jeu pour la reconnaissance de chiffres manuscrits. Nous disposons de centaines de photos en noir et blanc (de taille 28x28 pixels) de chiffres qui ont été étiquetés. Nous sommes donc dans le cas d'une classification multi-classes (10 classes : les chiffres 0, 1, ..., 9).

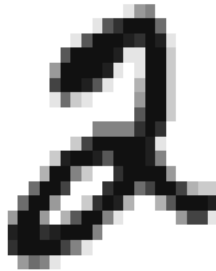


Figure 4: Une entrée du jeu de données MNIST.

## 2.2 Utilisation du package `nnet`

Ce package<sup>9</sup> permet la construction d'un perceptron à une couche cachée. Nous allons l'utiliser pour effectuer une classification (à 10 classes) i.e. pour créer un perceptron  $784-k-10$ , où  $k$  représente le nombre de neurones sur la couche cachée (et nous avons  $28 \times 28 = 784$ ). En effet, la plupart des logiciels de réseaux de neurones prennent en entrée un vecteur et non une matrice. Il faut donc en premier lieu **applatir** notre matrice pour créer un vecteur (qui contient les lignes de la matrice en file). Le package est assez simple d'utilisation (celui-ci ne contient que 6 méthodes).

On commence par normaliser nos entrées (les pixels ont une valeur entre 0 et 255). Cette étape est, comme l'applatissement de l'entrée, nécessaire pour l'utilisation du réseau.

```
train_x <- train$x/255
train_y <- train$y
test_x <- test$x/255
test_y <- test$y
```

La classification à plus de 2 classes contient une autre étape indispensable avant toute phase d'apprentissage d'un réseau neuronal : la conversion des étiquettes en ce qu'on appelle un *one hot encoded vector*. Cette étape s'effectue aisément avec la fonction `class.ind` du package :

```
train_y_cat <- class.ind(factor(train_y))
```

Puis on passe à la phase d'apprentissage. Ici, nous utiliserons dans un premier temps un réseau  $784-1-10$ , c'est-à-dire à un neurone dans la couche cachée. En classification, la fonction d'activation souvent utilisée pour la couche de sortie est la fonction softmax. Elle sort la probabilité d'être dans chaque classe, à partir d'entrées qui sont étalées sur  $\mathbb{R}$  tout entier. Celle-ci est généralement combinée avec une fonction de perte par entropie croisée. Par ailleurs, on fixe le nombre d'itérations maximal à 100 pour éviter un apprentissage trop long.

```
pmc <- nnet(x=train_x, y=train_y_cat, size=1, maxit=100,
```

---

<sup>9</sup><https://cran.r-project.org/web/packages/nnet/index.html>

```
softmax=TRUE)
```

Cette ligne s'est exécutée en `elapsed = 106s`. On obtient la précision suivante sur notre jeu de données test :

```
[1] "Accuracy: 0.2133"
```

Donc une bonne identification du chiffre dans seulement **21%** des cas. Cela reste mieux qu'une classification aléatoire (qui nous donnerait 10% de précision) mais c'est trop faible pour être industrialisé. La première idée serait d'augmenter le nombre de neurones sur la couche cachée, qui ici était fixé à 1. Malheureusement, `nnet` ne permet pas de faire cela, on obtient la sortie suivante :

```
Error in nnet.default(x = train_x, y = train_y_cat, size = 2, maxit = 100,  
  too many (1600) weights
```

Les 1600 poids correspondent aux  $784+784 w_i$  de la première couche auquel on ajoute deux biais  $b_i$ . Puis pour la couche de sortie :  $10+10 \tilde{w}_i$  et 10 biais  $\tilde{b}_i$ . On obtient un total de 1600.

**Remarque.** Ce package présente le désavantage d'avoir une documentation peu fournie et bien évidemment d'être limité dans son utilisation comme nous avons pu le voir.

## 2.3 Comparaison avec une autre méthode statistique : la forêt aléatoire

La forêt aléatoire<sup>10</sup> est une méthode ensembliste qui utilise plusieurs arbres de décisions afin de réduire la variance obtenue par un seul arbre. Il combine ces arbres dans une approche de type bagging.

Appliquons la méthode de la forêt aléatoire à notre problème de classification des chiffres manuscrits. On utilisera pour cette méthode le package R `randomForest`<sup>11</sup>. Celui-ci est très simple d'utilisation : il suffit de spécifier le nombre d'arbres de décision voulu. Voici un exemple.

```
numTrees <- 10  
rf <- randomForest(train_x, factor(train_y), xtest=test_x,  
  ytest=factor(test_y), ntree=numTrees)
```

Voici la sortie de ce petit bout de code :

Call:

```
randomForest(x = train_x, y = factor(train_y), xtest = test_x,  
  ytest = factor(test_y), ntree = numTrees)
```

---

<sup>10</sup>Voir [Ho, 1995]

<sup>11</sup><https://cran.r-project.org/web/packages/randomForest/index.html>

```

Type of random forest: classification
Number of trees: 10
No. of variables tried at each split: 28

```

Puis on obtient l'**erreur** et la **matrice de confusion** suivantes :

```

Test set error rate: 4.93%
Confusion matrix:
  0  1  2  3  4  5  6  7  8  9 class.error
0 967  0  1  1  1  3  2  1  4  0 0.01326531
1  0 1120  4  1  1  2  3  1  3  0 0.01321586
2  5  2 977  8  3  3  5  8 18  3 0.05329457
3  1  1 15 941  1 16  1 12 16  6 0.06831683
4  1  1  2  2 937  1  9  1  6 22 0.04582485
5  6  3  4 19  4 829 12  4  6  5 0.07062780
6  6  5  3  1  4  5 929  1  4  0 0.03027140
7  0  5 19  6  2  0  1 975  4 16 0.05155642
8  0  0  7 14  7 19 12  4 897 14 0.07905544
9  8  5  3  6 20 12  2  9  9 935 0.07333994

```

donc **95%** de bonne classification et cela pour une compilation d'environ deux minutes :

```

user  system elapsed
119.885  1.788 128.330

```

**Remarque.** L'augmentation du nombre d'arbres `numTrees` pourrait faire croître encore plus notre précision. Le temps de calcul serait naturellement augmenté.

Ainsi, on obtient des résultats largement supérieurs (95% contre 21% de justesse) à ceux obtenus avec le perceptron du package `nnet`. Cela est majoritairement dû au fait que notre réseau de neurones est trop basique. Pour complexifier celui-ci, nous devons faire appel à des logiciels plus sophistiqués. Cela fait l'objet de la section suivante.

### 3 Perceptron multi-couches avec le logiciel Tensorflow

Tensorflow est un logiciel développé par Google Brain qui est sorti sous licence Apache 2.0 en 2015. Il fait partie des logiciels les plus usités dans le domaine de l'apprentissage profond. Nous allons l'utiliser dans le même cadre que `nnet` sur le jeu de données de reconnaissance de chiffres MNIST.

**Remarque.** On utilisera le langage Python et la surcouche Keras pour une utilisation intuitive. On recommande l'utilisation de Google Colab pour des calculs sur carte graphique et donc beaucoup plus rapides. On rappelle que le code est disponible en entier en annexe, ici seuls les blocs fondamentaux sont présentés. On pourra se référer à l'annexe pour se donner une idée de la taille totale d'un script de PMC avec TensorFlow.

### 3.1 Construction de l'architecture

L'utilisation de Keras est assez simple. On commence par créer un modèle : ici on prends séquentiel, qui correspond à la structure du perceptron. Puis, on crée les couches de notre réseau une à une avec la fonction `add`, en spécifiant le type de la couche et son nombre de neurones. Ainsi, pour une couche complètement connectée on utilisera la fonction `Dense()` qui prends en paramètres son nombre de neurones et sa fonction d'activation. Les fonctions d'activations classiques sont pour la plupart implantées dans le logiciel. Par exemple, les fonctions d'activations énumérées au début de ce rapport y sont toutes intégrées.

Pour une couche qui transforme simplement une matrice en entrée en un vecteur (aplatissement de la matrice), on utilise la fonction `Flatten()`. Voici ce que cela donne pour un perceptron à deux couches cachées, chacune de 128 neurones :

PERCEPTRON MULTI-COUCHE AVEC KERAS : ARCHITECTURE.

---

```
model = Sequential()
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

---

**Remarque.** La couche `Dropout`<sup>12</sup> permet d'éviter le sur-apprentissage. Elle introduit du bruit dans l'apprentissage en supprimant aléatoirement des neurones de la couche précédente. Cette méthode permet souvent d'obtenir de meilleures performances de généralisation (score sur le jeu de test). Ici, le paramètre correspond à la proportion de neurones qui sont (temporairement) mis de côté.

### 3.2 Phase d'apprentissage

Nous avons l'architecture. Si l'on reprends les caractéristiques d'un réseau de neurones (cf partie 1.4), il nous faut maintenant préciser la fonction de coût, la méthode d'optimisation et la métrique. Cela s'effectue avec la fonction `compile`. Puis, on effectue la phase d'apprentissage avec la fonction `fit` en précisant la taille des lots (`batch_size`) et le nombre d'époque. Finalement, on utilise `evaluate` pour obtenir l'évaluation de la métrique sur notre estimateur  $\hat{y}$ .

PERCEPTRON MULTI-COUCHE AVEC KERAS : APPRENTISSAGE.

---

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
```

---

<sup>12</sup>[Srivastava et al., 2014]



```

        batch_size=batch_size,
        epochs=epochs,
        verbose=1,
        validation_data=(x_test, y_test))

score = model.evaluate(x_test, y_test, verbose=0)

```

---

**Remarque.** Le paramètre `verbose` gère l’affichage des messages par le logiciel dans le terminal. 0 signifiant le minimum de messages, 1 tous les messages.

Ce très court bout de code nous permet d’obtenir une classification précise à **97.4%**. En effet, voici la fin de la sortie :

```

Epoch 12/12
60000/60000 [=====] - 13s 209us/step -
loss: 0.1391 - accuracy: 0.9568 - val_loss: 0.0877 - val_accuracy: 0.9739
Test loss: 0.08773939539240673
Test accuracy: 0.9739000201225281
CPU times: user 1min 20s, sys: 34.5 s, total: 1min 55s
Wall time: 2min 40s

```

**Remarque.** L’utilisation de TensorFlow est rendu pratique par la large documentation qui l’accompagne ainsi que son utilisation très répandue dans le milieu de l’apprentissage automatique. Par ailleurs, le logiciel contient les meilleures implémentations des méthodes d’optimisation courantes et il est régulièrement mis à jour par les équipes de Google. Nous utilisons ici la version tf 1. Cependant, la version tf 2 est maintenant la version de référence. A savoir qu’il existe aussi le logiciel PyTorch qui effectue peu ou prou la même chose (PyTorch est développé par les équipes de Facebook AI Research).

## 4 Carte de Kohonen (ou carte auto-adaptative)

### 4.1 Principe

Les cartes de Kohonen<sup>13</sup> sont un type de réseau de neurones utilisés en apprentissage non supervisé. Celles-ci ont pour objectif de réduire la dimension de notre entrée  $x$  (comme l’analyse en composantes principales par exemple). Elles sont très utiles pour la visualisation et l’analyse de **données de grande dimension**.

Le principe est le suivant : on dispose de neurones disposés sur un maillage. Chaque neurone de ce maillage est connecté à tous les neurones en entrée (les  $x_1, \dots, x_n$ ) et correspond à un vecteur poids  $(w_i)_{1 \leq i \leq N^2}$  de même dimension que les  $x_i$  ( $N \times N$  est la taille de notre maillage, celui-ci est à fixer). La carte de Kohonen va établir des groupements dans lesquels les vecteurs seront proches au sens de la distance choisie.

La phase d’apprentissage se déroule ainsi :

---

<sup>13</sup>[Kohonen, 1982], [Kohonen, 2001]

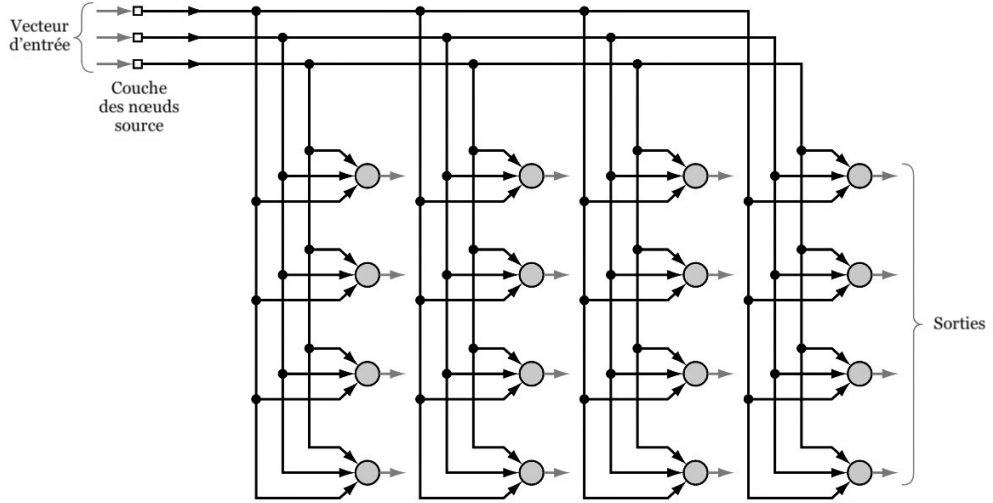


Figure 5: Architecture d’une carte de Kohonen. Cas d’un maillage  $4 \times 4$  avec une entrée tri-dimensionnelle. Traduction d’une illustration de [Haykin, 2008].

- Initialiser les poids  $\mathbf{w}_i$ . Initialiser  $t = 0$ .
- Tirer une observation aléatoire  $\mathbf{x}$  dans le jeu.
- Déterminer le neurone  $I$  le plus proche de cette observation (appelé le BMU pour *best matching unit* ou parfois aussi le neurone gagnant) :  $\mathbf{w}_I = \operatorname{argmin}_i \|\mathbf{x} - \mathbf{w}_i\|$ .
- Approcher le neurone  $I$  de l’observation. Les voisins du BMU sont aussi rapprochés dans une moindre mesure. Concrètement,  $\forall i = 1, \dots, N^2$  :

$$\mathbf{w}_i = \mathbf{w}_i + \eta(t) h_I(i) (\mathbf{x} - \mathbf{w}_i)$$

avec  $\eta$  le pas (décroissant) et  $h$  la fonction voisinage (attention à ne pas confondre avec nos notations des sections précédentes où  $h$  était la sortie d’un neurone). La fonction voisinage est à choisir. Celle-ci doit avoir une valeur élevée au point  $I$  et décroître au fur et à mesure que l’on en s’éloigne sur le maillage. Par exemple, une gaussienne centrée en  $I$ .

- $t = t + 1$
- Tant que  $t < t_{\max}$ , reprendre depuis l’étape 2 (où  $t_{\max}$  représente le nombre d’itérations voulu).

**Remarque.** Plusieurs façon d’initialiser les poids  $\mathbf{w}_i$  existent. Par exemple, initialiser sur des points tirés aléatoirement dans le jeu de données.

**Remarque.** Il existe plusieurs bibliothèques en Python qui permettent de représenter la carte de Kohonen associée à un jeu de données : `minisom`, `sompy`.

## 4.2 Utilisation pratique

Nous utiliserons le package `MiniSom`<sup>14</sup> qui se base sur l'architecture de `NumPy`. Celui-ci est assez complet, comprenant notamment plusieurs fonctions voisinage et méthodes d'initialisation. Par ailleurs, on utilisera le jeu de données MNIST pour pouvoir comparer nos résultats avec ceux obtenus dans les sections précédentes.

Le pré-traitement à appliquer est le même que pour la plupart des algorithmes d'apprentissage automatique : normalisation des données. Ici, cela est d'autant plus nécessaire que nous appliquerons une analyse en composantes principales (ACP) pour initialiser les poids  $w_i$ . Ici, nos données sont sous la forme d'un tenseur (matrice tri-dimensionnelle)  $60000 \times 28 \times 28$ . Pour utiliser `minisom`, nous devons les aplatir afin d'obtenir une matrice bi-dimensionnelle  $60000 \times 784$ .

Une fois le pré-traitement des données effectués, le logiciel s'utilise aisément :

CARTE DE KOHONEN AVEC MINISOM

---

```
from minisom import MiniSom
som = MiniSom(30, 30, 784,
              learning_rate=0.5,
              neighborhood_function='triangle')
som.pca_weights_init(x_train)
som.train_random(x_train, 5000, verbose=True)
```

---

Les deux premiers paramètres correspondent à la taille du maillage. Le suivant à la taille de nos observations. Puis on choisit un taux de décroissance du pas ainsi que la fonction de voisinage. D'autres fonctions de voisinage que `triangle` sont utilisables : `gaussienne`, `bulle`, `chapeau mexicain`. L'initialisation des poids  $w_i$  est faite après avoir effectué une ACP et de manière à couvrir les deux premières composantes principales. Cela permet une convergence plus rapide. Puis, on effectue un apprentissage en tirant aléatoirement des observations via la fonction `train_random`. On spécifie aussi le nombre d'itérations. La sortie est la suivante :

```
quantization error: 0.26291870545452456
topographic error: 0.1605
```

Nous avons notre carte de taille  $30 \times 30$ . Nous pouvons pour chaque neurone déterminer l'observation la plus proche et afficher cette carte des observations. La figure 6 représente une partie de taille  $20 \times 20$  de la carte des observations lié à notre carte de Kohonen.

**Observations.** On observe que certains "4" sont très proches de "9", d'où la proximité sur notre maillage. Par ailleurs, quelques frontières se dessinent (zones blanches). Une telle carte permet également de faire de la prédiction. Etant donnée une nouvelle observation (image d'un chiffre), on peut calculer son neurone le plus proche sur la carte. Par exemple, si l'on obtient le résultat présenté en figure 7, on est assez confiant sur la sortie liée à cette nouvelle observation. En effet, le neurone choisi est un 8 qui est lui-même entouré uniquement de 8. On conclut assez sereinement que la nouvelle entrée est un 8.

---

<sup>14</sup>MiniSom par Giuseppe Vettigli

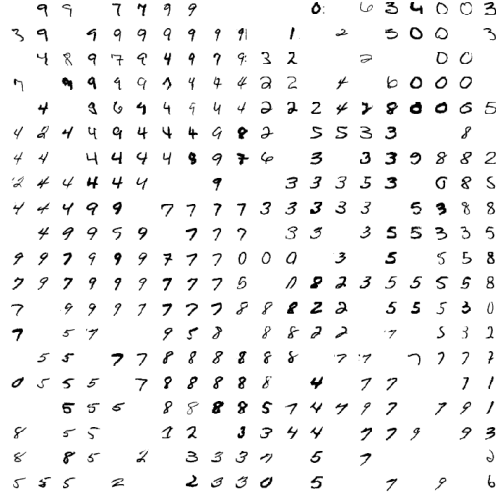


Figure 6: Partie d’une carte de Kohonen sur le jeu de données MNIST.



Figure 7: Détermination du neurone le plus proche pour une nouvelle observation.

## 5 Machine de Boltzmann restreinte

### 5.1 Présentation

La machine de Boltzmann restreinte<sup>15</sup> est une méthode d’apprentissage non-supervisé. C’est un réseau de neurones composé de deux couches complètement connectées de manière symétriques. Une couche est appelée visible et l’autre cachée (ou latente). Ainsi, chaque neurone de la couche visible est connecté à tous les neurones de la couche cachée, et inversement. C’est pourquoi la machine est dite **restreinte** : nous avons réduit le nombre de connexions dans le réseau comme on peut le voir en figure 8. Par ailleurs, dans sa version de référence, les neurones sont des Bernoulli : ils ne sont pas déterministes mais ce sont des **variables aléatoires**. On parle aussi de réseau neuronal stochastique. On remarque notamment qu’il n’y a pas de neurones de sortie.

La machine de Boltzmann restreinte va générer une estimation de la densité des entrées  $x$ .

**Definition.** Soit  $x \in \{0, 1\}^d$ . On note  $E$  la fonction d’énergie et on a

$$E(x) := -x^T U x - b^T x$$

<sup>15</sup>[Smolensky, 1986]

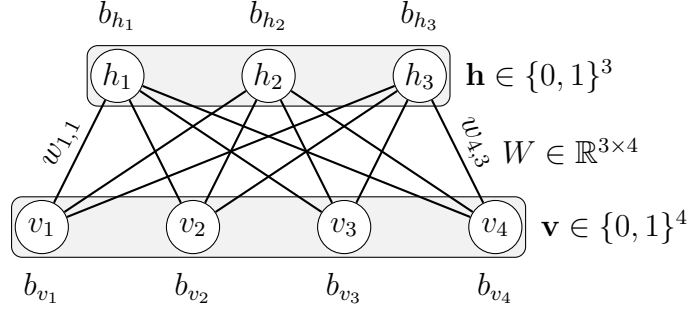


Figure 8: Structure d'une machine de Boltzmann restreinte

avec  $U$  la matrice des poids et  $b$  le vecteur des biais. Nous avons la loi jointe de  $X_1, \dots, X_n$  :

$$p(\mathbf{x}, \theta) = \frac{e^{-E(\mathbf{x}, \theta)}}{Z(\theta)}$$

avec  $Z$  la fonction de normalisation (aussi appelée fonction de partition).

**Remarque.** C'est un estimateur universel de densité de variables aléatoires discrètes. Voir [Le Roux & Bengio, 2008].

On décompose alors  $\mathbf{x}$  en  $\mathbf{v}$  les unités visibles et  $\mathbf{h}$  les unités cachées (ou latentes). L'énergie devient  $E(\mathbf{v}, \mathbf{h}) = -\mathbf{v}^T R \mathbf{v} - \mathbf{v}^T W \mathbf{h} - \mathbf{h}^T S \mathbf{h} - \mathbf{b}^T \mathbf{v} - \mathbf{c}^T \mathbf{h}$ .

**Remarque.** Nous n'aurons pas de formule close pour  $Z = \sum_{\mathbf{v}} \sum_{\mathbf{h}} E(\mathbf{v}, \mathbf{h})$  la fonction de partition. Voir [Long & Servedio, 2010].

$p$  n'est pas non plus calculable. On passe par  $p(\mathbf{h}|\mathbf{v})$  et  $p(\mathbf{v}|\mathbf{h})$ . Nous avons

$$p(V = \mathbf{v}, H = \mathbf{h}) = \frac{1}{Z} e^{-E(\mathbf{v}, \mathbf{h})}$$

où  $E(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^T \mathbf{v} - \mathbf{c}^T \mathbf{h} - \mathbf{v}^T W \mathbf{h}$ .

$$\begin{aligned} p(\mathbf{h}|\mathbf{v}) &= \frac{p(\mathbf{h}, \mathbf{v})}{p(\mathbf{v})} \\ &= \frac{1}{Z p(\mathbf{v})} \exp(\mathbf{b}^T \mathbf{v} + \mathbf{c}^T \mathbf{h} + \mathbf{v}^T W \mathbf{h}) \\ &= \frac{1}{\tilde{Z}} \exp(\mathbf{c}^T \mathbf{h} + \mathbf{v}^T W \mathbf{h}) \\ &= \frac{1}{\tilde{Z}} \end{aligned}$$

En particulier

$$\begin{aligned} p(h_j = 1|\mathbf{v}) &= \frac{\tilde{p}(h_j = 1|\mathbf{v})}{\tilde{p}(h_j = 1|\mathbf{v}) + \tilde{p}(h_j = 0|\mathbf{v})} \\ &= \frac{\exp(c_j + \mathbf{v}^T W_{:,j})}{\exp(0) + \exp(c_j + \mathbf{v}^T W_{:,j})} \end{aligned}$$

et donc

$$p(h_j = 1|\mathbf{v}) = \sigma(c_j + \mathbf{v}^T W_{:,j})$$

avec  $\sigma$  la fonction sigmoïde. Par un raisonnement analogue, on a

$$p(v_i = 1|\mathbf{h}) = \sigma(b_i + \mathbf{h}^T W_{:,i})$$

**Remarque.** La couche visible contient autant de neurones qu'il y a d'entrées  $x_1, \dots, x_n$ .

## 5.2 Apprentissage et divergence contrastive

La machine de Boltzmann restreinte cherche à approximer le maximum de vraisemblance par l'algorithme de divergence contrastive. Cette méthode a été proposée par Geoffrey Hinton<sup>16</sup>. Nous utilisons l'échantillonnage de Gibbs (qui est une méthode MCMC) pour approximer une espérance.

Avec les notations utilisées dans cette sous-partie, nous avons

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{b}_v^T \mathbf{v} - \mathbf{b}_h^T \mathbf{h} - \mathbf{v}^T W \mathbf{h}$$

Puis nous avons la log-vraisemblance des données observées (ou visibles)

$$\ell(W) = \log \prod_{\mathbf{v}} p(V = \mathbf{v}) = \sum_{\mathbf{v}} \log p(V = \mathbf{v})$$

On insère dans cette équation les expressions suivantes :

$$\begin{cases} p(V = \mathbf{v}) = \frac{1}{Z} \sum_{\mathbf{h}} \exp(-E(\mathbf{v}, \mathbf{h})) \\ Z = \sum_{\mathbf{v}, \mathbf{h}} \exp(-E(\mathbf{v}, \mathbf{h})) \end{cases}$$

et on obtient

$$\ell(W) = \sum_{\mathbf{v}} \log \sum_{\mathbf{h}} \exp(-E(\mathbf{v}, \mathbf{h})) - \log \sum_{\mathbf{v}', \mathbf{h}'} \exp(-E(\mathbf{v}', \mathbf{h}'))$$

On dérive par rapport aux poids :

$$\begin{aligned} \frac{\partial \ell(W)}{\partial w_{i,j}} &= \frac{1}{n} \sum_{\mathbf{v}} \left( - \sum_{\mathbf{h}} p(H = \mathbf{h} | V = \mathbf{v}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial w_{i,j}} + \sum_{\mathbf{v}, \mathbf{h}} p(V = \mathbf{v}, H = \mathbf{h}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial w_{i,j}} \right) \\ &= \frac{1}{n} \sum_{\mathbf{v}} \left( \sum_{\mathbf{h}} p(H = \mathbf{h} | V = \mathbf{v}) h_i v_j - \sum_{\mathbf{v}} p(V = \mathbf{v}) \sum_{\mathbf{h}} p(H = \mathbf{h} | V = \mathbf{v}) h_i v_j \right) \end{aligned}$$

Pour simplifier, on introduit la notation suivante<sup>17</sup> :

$$\langle v_i h_j \rangle_{data} := \langle v_i h_j \rangle_{p(\mathbf{h}|\mathbf{v})q(\mathbf{v})} = \frac{1}{n} \sum_{\mathbf{v}} \sum_{\mathbf{h}} v_i h_j p(H = \mathbf{h} | V = \mathbf{v})$$

<sup>16</sup>Voir [Hinton, 2002]. A noter que d'autres méthodes que la divergence contrastive peuvent être utilisées. Celle-ci est cependant la plus courante.

<sup>17</sup>Même notation que dans : A Practical Guide to Training RBMs (Hinton, 2010)

avec  $q$  la loi empirique des données et :

$$\langle v_i h_j \rangle_{model} := \langle v_i h_j \rangle_{p(\mathbf{v}, \mathbf{h})} = \sum_{\mathbf{v}} \sum_{\mathbf{h}} v_i h_j p(V = \mathbf{v}, H = \mathbf{h})$$

On a donc avec ces notations :

$$\boxed{\frac{\partial \ell(W)}{\partial w_{i,j}} = \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}}$$

La mise à jour des poids de la matrice s'effectuera par ascension stochastique de plus grande pente :

$$w_{i,j}(t+1) = w_{i,j}(t) + \alpha(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model})$$

où  $\alpha$  est le pas d'apprentissage. On trouve par des raisonnements similaires :

$$\frac{\partial \ell(W)}{\partial (b_v)_j} = v_j - \sum_{\mathbf{v}} p(\mathbf{v}) v_j$$

et pour le biais  $b_h$  des unités cachées :

$$\frac{\partial \ell(W)}{\partial (b_h)_i} = p(h_i = 1 | \mathbf{v}) - \sum_{\mathbf{v}} p(\mathbf{v}) p(h_i = 1 | \mathbf{v})$$

Il est facile de calculer  $\langle v_i h_j \rangle_{data}$  car nous avons une formule close comme nous l'avons vu précédemment. Cependant, il est moins aisé de calculer  $\langle v_i h_j \rangle_{model}$ .

**Remarque.** La mise à jour d'un poids  $w_{i,j}$  reliant deux unités ne s'effectue qu'à partir de la connaissance de la statistique de ces deux unités. Nous n'avons besoin d'aucune connaissance sur le reste du réseau ou encore sur la manière dont les deux statistiques ont été produites. La règle d'apprentissage est dite **locale**.

### Algorithme de divergence contrastive à $k$ itérations.

1ERE ÉTAPE. On applique à chaque entrée la matrice des poids et le biais de la couche cachée.  $h^{(n+1)} = 1$  avec probabilité  $\sigma(v^{(n)T}W + b_h)$  où  $b_h = (b_{h_1}, \dots, b_{h_d})^T$  le biais avec  $d$  le nombre d'unités cachées. La fonction d'activation étant le sigmoïde, nous avons bien  $\sigma(v^{(n)T}W + b_h) \in (0, 1)$ . En pratique, on tire une uniforme  $U$  entre 0 et 1. Si  $\sigma(v^{(n)T}W + b_h) > U$  alors  $h^{(n+1)} = 1$ , sinon  $h^{(n+1)} = 0$ . Il est important que nos unités restent binaires et ne soient pas des probabilités. En fait, cela permet de créer un régularisateur. Comme nous allons le voir, cette étape ainsi que la suivante peuvent être répétées plusieurs fois. Lors de la dernière itération, nous garderons les unités cachées sous forme de probabilité. Cela permet un apprentissage légèrement plus rapide.

2EME ÉTAPE. On effectue le chemin dans l'autre sens. C'est la **phase de reconstruction**. Les unités visibles sont mises à jour :

$$v^{(n+1)} = \sigma(h^{(n+1)T}W + b_v)$$

où  $b_v = (b_{v_1}, \dots, b_{v_n})^T$  est le biais de la couche visible. Ici, nous n'avons pas besoin d'utiliser des unités binaires lors des itérations comme c'est le cas avec les unités cachées.

**3EME ÉTAPE.** Les deux étapes précédentes représentent une itération d'**échantillonnage de Gibbs** (alterné). Elles peuvent être répétées  $k$  fois et on les dénote par  $CD_k$  (pour *contrastive divergence*). Pour  $k \rightarrow \infty$ , nous avons  $(v^{(k)}, h^{(k)})$  qui sont des observations de loi  $p(v, h)$ . La matrice des poids  $W$  est mise à jour seulement après avoir effectué  $CD_k$ . On applique maintenant la règle d'apprentissage :

$$w_{i,j}(t+1) = w_{i,j}(t) + \alpha(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model})$$

autrement dit

$$w_{i,j}(t+1) = w_{i,j}(t) + \alpha \left( p(h_i = 1 | \mathbf{v}^{(0)}) \cdot v_j^{(0)} - p(h_i = 1 | \mathbf{v}^{(k)}) \cdot v_j^{(k)} \right)$$

et on met à jour les biais de la manière suivante :

$$\mathbf{b}_v(t+1) = \mathbf{b}_v(t) + \alpha \left( \mathbf{v}^{(0)} - \mathbf{v}^{(k)} \right)$$

$$\mathbf{b}_h(t+1) = \mathbf{b}_h(t) + \alpha \left( p(\mathbf{h} = 1 | \mathbf{v}^{(0)}) - p(\mathbf{h} = 1 | \mathbf{v}^{(k)}) \right)$$

Avec cette règle nous approchons le gradient d'une fonction objectif appelée la divergence contrastive. C'est la différence entre deux divergences de Kullback-Leibler.<sup>18</sup>

**Remarque.** On a la convergence pour  $CD_1$ .

**Remarque.** Les machines de Boltzmann restreintes sont de moins en moins utilisées et la communauté de l'apprentissage profond privilégie maintenant les réseaux antagonistes génératifs (GAN) ou les auto-encodeurs.

### 5.3 Application à la classification

Une machine de Boltzmann restreinte peut être utilisée pour différentes tâches. On peut par exemple entraîner notre machine sur un jeu de données. Celle-ci va estimer la distribution de nos données. Puis, on peut demander à notre machine de générer de nouveaux exemples à partir de la distribution estimée. On peut aussi effectuer de la classification. En effet, on entraîne notre RBM avec les données d'entrée  $x$  et de sortie  $y$ . La machine va estimer la distribution jointe de  $(x, y)$ , ceux-ci étant les neurones visibles. Puis, on donne à la machine une nouvelle entrée et elle va nous générer une sortie correspondante.

Par ailleurs, les RBM ont été majoritairement utilisés pour faire du pré-traitement avant application d'une méthode de classification. Par exemple, on peut utiliser un RBM sur nos données de chiffres manuscrits pour faire de l'apprentissage non-supervisé. Puis à partir des caractéristiques extraites, appliquer une méthode d'apprentissage supervisé comme un

---

<sup>18</sup>Pour être tout à fait précis, nous approchons le gradient d'aucune fonction comme l'on montré [Sutskever & Tieleman, 2010]. Cependant, cette méthode a de bons résultats empiriques.



perceptron multi-couches par exemple. Nous utiliserons la bibliothèque scikit-learn et sa classe `neural_network.BernoulliRBM`.

Pour des raisons de puissance de calcul insuffisante, nous n'avons pu aboutir à de bons résultats pour notre jeu de données MNIST (le code est disponible en annexe). Le lecteur trouvera un programme similaire sur la documentation de scikit-learn : Restricted Boltzmann Machine features for digit classification. Une comparaison est effectuée entre une régression logistique appliquée directement sur un jeu de données de chiffres manuscrits et une régression logistique appliquée au même jeu mais qui a subi un pré-traitement à l'aide d'une machine de Boltzmann restreinte. On observe une précision bien supérieure dans ce deuxième cas : 94% contre 78%.

## 6 Réseaux de neurones convolutionnels

### 6.1 Principe

Un réseau convolutionnel est un réseau de neurones qui comprends au moins une couche de convolution. Une couche de convolution est composée de trois étapes :

- Opération de convolution (il s'agit d'une transformation affine).
- Détecteur : application d'une fonction non linéaire (nos fonctions d'activations classique, par exemple le ReLU).
- Opération dite de *pooling* (groupage).

Nous allons donc détailler le principe des deux nouvelles opérations (par rapport à un perceptron).

#### 6.1.1 Convolution

Dans le cadre de l'apprentissage profond, nous utilisons la mesure de comptage et les notations suivantes :

$$(x * w)(t) = \sum_{n=-\infty}^{+\infty} x(n)w(t-n)$$

où  $x$  est l'entrée et  $w$  le noyau (ou le filtre).

Pour une **image**  $I$  de taille  $M \times N$  et un noyau  $K$ , cela donne

$$(I * K)(i, j) = \sum_{m=1}^M \sum_{n=1}^N I(i-m, j-n)K(m, n)$$

où on a utilisé la définition et la propriété de commutativité. En pratique, toujours avec notre exemple d'une image, le filtre  $K$  correspond à une petite matrice que l'on va multiplier à toutes les régions de même taille de notre entrée, avec un pas  $s$  à fixer à l'avance. Il s'agit

donc de faire glisser un patron le long de la première ligne avec un pas  $s$ , puis de passer à la deuxième ligne, etc. Il peut être utile d'ajouter un cadre de 0 autour de notre matrice de pixels pour ne pas réduire la dimension de la sortie. Lorsque cela n'est pas effectué et que la patron est appliqué seulement lorsqu'il est entièrement contenu dans l'image, on parle parfois de **convolution valide** (la sortie a une taille strictement inférieure à l'entrée).

Dans un réseau convolutionnel, le noyau  $K$  est un hyper-paramètre : il est modifié durant la phase d'apprentissage. Selon le noyau  $K$ , le réseau extrait des informations différentes, comme les contours des objets, les couleurs, la forme d'un œil.

**Remarque.** L'opération de convolution peut être écrite sous la forme d'une **transformation affine** (produit d'une matrice avec l'entrée  $\mathbf{x}$ ). Pour des raisons pratiques (réduction du temps de calcul par parallélisation des opérations), il est bien plus efficace d'effectuer la convolution par l'opération décrite ci-dessus de glissement d'un filtre le long de l'entrée.

### 6.1.2 Groupage (ou pooling)

Le *pooling* consiste à remplacer un neurone de la sortie par un "résumé" de son voisinage (neurone compris). Par exemple, on peut prendre le maximum d'une petite région autour de l'unité en question (ou bien une moyenne pondérée, la norme  $L^2$ , etc.). Ainsi, la position des neurones joue un rôle dans les réseaux convolutionnels, contrairement aux PMC par exemple.

Pour une image, cela consiste à passer un masque de largeur  $l$  sur la matrice de pixels, avec un pas  $s$ . Selon la largeur et le pas, cela peut mener à une réduction de dimension de l'entrée comme cela est illustré sur la figure 9. On y observe une réduction par  $2^2$  de la dimension de l'entrée.

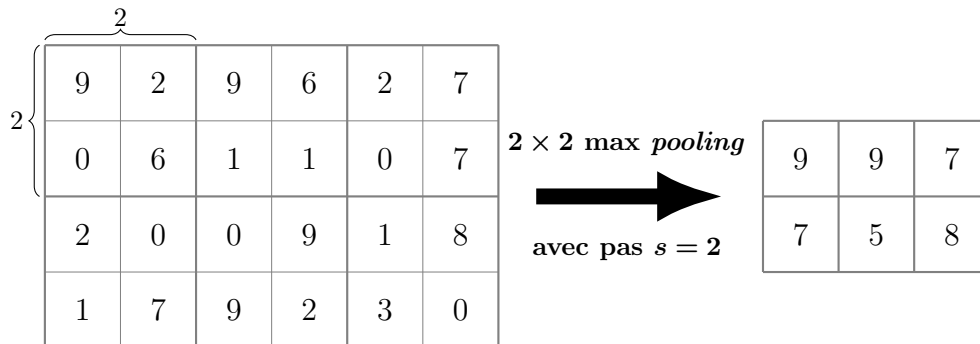


Figure 9: Opération de *pooling* avec réduction de dimension (taille  $2 \times 2$  et pas de 2).

La procédure de *pooling* permet de créer une représentation approximativement invariante par translation de l'entrée (petites translations). Une translation de toute une image de 1 pixel sur la droite ne change pas fondamentalement l'information transmise par l'image. Le pooling permet de passer outre ces petites transformations. Par ailleurs, il est possible de créer un réseau convolutionnel invariant à d'autres transformations, comme des rotations d'image<sup>19</sup>.

<sup>19</sup>Voir [Goodfellow et al., 2013].

## 6.2 Utilisation pratique d'un réseau convolutionnel

Un réseau convolutionnel permet d'effectuer des tâches de classification et de régression. Nous allons utiliser un réseau convolutionnel pour traiter notre jeu de données MNIST et ce par le biais du logiciel Tensorflow, comme précédemment avec le perceptron multi-couches. Le code va d'ailleurs en être très proche. Nous n'ajouterons qu'à ajouter deux couches de convolution au début du réseau.

**Remarque.** Un réseau convolutionnel permet de traiter des entrées à taille variable (non connue à l'avance). Cela représente un net avantage face aux réseaux traditionnels comme le perceptron multi-couche. Il peut donc être intéressant, même lorsque notre méthode classique fonctionne bien, d'utiliser un réseau convolutionnel afin d'obtenir un algorithme plus général qui traite des entrées de tailles variées. Cela est souvent nécessaire en traitement d'images.

Pour traiter notre jeu contenant des image (donc des entrées de dimension 2), nous utiliserons la fonction `keras.layers.Conv2D()`. Par ailleurs, nous effectuerons un groupage par maximum avec la fonction `keras.layers.MaxPooling2D()`.

RÉSEAU CONVOLUTIONNEL AVEC KERAS : ARCHITECTURE.

---

```
model = Sequential()
model.add(Conv2D(filters=32,                                #output size
                 kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape,
                 padding='valid',                            #default
                 stride=(1,1)))                             #default
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2),
                      strides=None,                          #use pool_size
                      padding='valid'))                       #default
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

---

Ici, nous avons introduit deux nouveaux type de couches par rapport à notre perceptron multi-couches. D'une part, la couche de convolution `Conv2D`. Les paramètres parlent d'eux-mêmes. Le `padding` permet de spécifier si l'on souhaite effectuer une convolution valide ou non. Pour une convolution qui préserve la dimension, on utilisera `padding='same'`. Le `stride` correspond à notre pas  $s$  (une coordonnée par dimension). Le paramètre `input_shape` est nécessaire pour la première couche du réseau convolutionnel. D'autre part, nous avons la couche de groupage `MaxPooling2D` qui prends en paramètres la taille de groupage, le pas (ici, on prends le pas par défaut qui est la taille de groupage), ainsi que le type de convolution : valide ou non (ici c'est donc un type de groupage, le principe est le même qu'avec le type de convolution). Les autres types de couches ont déjà été abordés dans la section 3.

Nous avons donc effectuer deux convolutions successives puis un groupage. Enfin, nous avons ajouter une couche d'aplatissement `Flatten` qui nous fournit un vecteur que l'on jette dans un perceptron. Nous avons construit notre architecture de réseau convolutionnel. Passons à la phase d'apprentissage.

RÉSEAU CONVOLUTIONNEL AVEC KERAS : PHASE D'APPRENTISSAGE.

---

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

```
model.fit(x_train, y_train,
         batch_size=batch_size,
         epochs=epochs,
         verbose=1,
         validation_data=(x_test, y_test))
```

```
score = model.evaluate(x_test, y_test, verbose=0)
```

---

Nous obtenons alors la sortie suivante :

```
Epoch 1/12
60000/60000 [=====] - 19s 321us/step -
loss: 0.2573 - acc: 0.9217 - val_loss: 0.0535 - val_acc: 0.9825
Epoch 2/12
60000/60000 [=====] - 6s 102us/step -
loss: 0.0845 - acc: 0.9748 - val_loss: 0.0461 - val_acc: 0.9846
Epoch 3/12
60000/60000 [=====] - 6s 101us/step -
loss: 0.0650 - acc: 0.9806 - val_loss: 0.0334 - val_acc: 0.9888
Epoch 4/12
60000/60000 [=====] - 6s 101us/step -
loss: 0.0523 - acc: 0.9843 - val_loss: 0.0319 - val_acc: 0.9892
Epoch 5/12
60000/60000 [=====] - 6s 101us/step -
loss: 0.0451 - acc: 0.9860 - val_loss: 0.0325 - val_acc: 0.9888
Epoch 6/12
60000/60000 [=====] - 6s 100us/step -
loss: 0.0405 - acc: 0.9873 - val_loss: 0.0284 - val_acc: 0.9914
Epoch 7/12
60000/60000 [=====] - 6s 102us/step -
loss: 0.0368 - acc: 0.9892 - val_loss: 0.0301 - val_acc: 0.9901
Epoch 8/12
60000/60000 [=====] - 6s 99us/step -
loss: 0.0348 - acc: 0.9894 - val_loss: 0.0306 - val_acc: 0.9899
Epoch 9/12
```

```

60000/60000 [=====] - 6s 101us/step -
loss: 0.0318 - acc: 0.9903 - val_loss: 0.0267 - val_acc: 0.9910
Epoch 10/12
60000/60000 [=====] - 6s 99us/step -
loss: 0.0280 - acc: 0.9913 - val_loss: 0.0299 - val_acc: 0.9901
Epoch 11/12
60000/60000 [=====] - 6s 101us/step -
loss: 0.0266 - acc: 0.9918 - val_loss: 0.0268 - val_acc: 0.9906
Epoch 12/12
60000/60000 [=====] - 6s 101us/step -
loss: 0.0263 - acc: 0.9913 - val_loss: 0.0270 - val_acc: 0.9914
Test loss: 0.02704868172882725
Test accuracy: 0.9914

```

Nous avons donc une justesse de **99.14%**. Notre précision est donc meilleure qu'avec le perceptron multi-couche de la section 3 qui atteignait une justesse de 97.4%. On peut observer sur la figure 10 les images qui ont mal été classés par notre réseau convolutionnel.

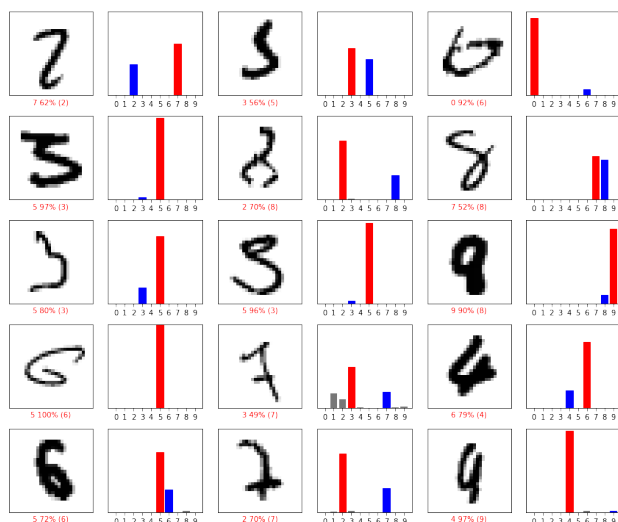


Figure 10: Quelques mauvaises classifications de notre réseau convolutionnel sur le jeu MNIST (les mauvaises classifications représentent 1% du jeu).

On remarque notamment sur la figure 10 que certains chiffres sont tout à fait lisibles. Même un observateur ne connaissant pas les chiffres arabes pourrait comparer ces entrées au reste du jeu et facilement en déduire leur étiquette associée. Il est donc assez difficile d'interpréter à partir de ces exemples quelles sont les difficultés que le réseau rencontre. Un autre fait remarquable est la certitude que le réseau peut avoir sur certains exemples, comme pour le 6 de l'avant dernière ligne qu'il classe en tant que 5 avec une certitude de 100% (la certitude correspond aux valeurs du vecteur de la couche de sortie softmax, ici elles valent 0 sauf en septième position où on a un 1).

Il est donc souvent intéressant d'utiliser les réseaux convolutionnels dans des approches de type *bagging*, en le combinant avec d'autres algorithmes très puissants comme le *Gradient*

*Boosting*. Par ailleurs, le modèle utilisé ici pourrait sûrement être amélioré par des méthodes de validation croisée (*k-fold cross validation*) ou de recherche sur grille (*grid-search*) des hyper-paramètres optimaux. Cependant, ces pistes nécessitent une puissance et un temps de calcul plus élevés.

## 7 Conclusion du rapport

Les réseaux de neurones artificiels représentent un domaine assez vaste de l'apprentissage automatique. Comme nous avons pu le voir, il est possible de créer des méthodes très variées dans leur fonctionnement intrinsèque. Ces méthodes ont des applications différentes : apprentissage supervisé, non-supervisé, classification, régression et ce dans des domaines variés (vision artificielle, maintenance prédictive, etc.). Ces méthodes **évoluent constamment** et chaque année de nouveaux algorithmes remplacent les anciens. Par exemple, les machines de Boltmann restreintes laissent désormais place aux réseaux dits antagonistes génératifs.

Par ailleurs, ce domaine est extrêmement lié à l'évolution des **puissances de calcul** des ordinateurs ainsi que des logiciels (**gestion de la mémoire**, etc.). Ainsi, on observe que le perceptron multi-couches est utilisé avec des architectures très profondes, ce qui n'était pas possible il y a quelques années du fait du nombre de connexions élevé, le réseau étant souvent dense. Un réseau aussi basique que le perceptron trouve alors des performances très bonnes sur une grande variété d'applications.

Il est parfois assez difficile de comprendre pourquoi un réseau profond a de bonnes performances sur tel jeu de données et de moins bonnes performances sur tel autre jeu de données. Par exemple, un réseau convolutionnel en traitement d'image va après chaque couche extraire des informations sur l'image : formes, contour d'un élément particulier. Mais il n'est pas aisé de comprendre pourquoi certaines images posent problème au réseau. En effet, comme nous l'avons vu sur le jeu MNIST, certains chiffres manuscrits très similaires à leurs homologues ne sont pas reconnus par le réseau convolutionnel. Par ailleurs, il est possible de brouiller une image de manière à ce qu'elle ne soit pas comprise par un réseau convolutionnel mais que pour un observateur celle-ci n'ait pas changé. Comme avec toute méthode statistique, la phase de pré-traitement des données reste fondamentale.

On observe dans le domaine de l'apprentissage profond que de nombreuses méthodes ont des **résultats remarquables** en pratique mais que leur **fondement théorique** est très limité, la pratique évoluant très rapidement.

Les applications qui ont été faites ici avec le jeu de données MNIST peuvent être facilement faites avec d'autres jeux, comme par exemple ImageNet ou CIFAR-10 (mais aussi des jeux qui ne contiennent pas d'images dans une certaine mesure). Une fois la partie de pré-traitement effectuée, le code est quasiment **ré-utilisable** dans sa forme actuelle. Ici nous avons fait le choix de prendre un jeu assez simple pour que les calculs se fassent assez rapidement.

## A Code des script R et Python

### A.1 Perceptron multi-couches sous R et comparaison avec l'algorithme forêt aléatoire

```
# author = Raphael Mignot
# date = March 2020

load_mnist <- function() {
  # Function needed to load the mnist dataset from its
  # original source : http://yann.lecun.com/exdb/mnist/
  # Source code : brendan o'connor -
  # gist.github.com/39760 - anyall.org
  load_image_file <- function(filename) {
    ret = list()
    f = file(filename, 'rb')
    readBin(f, 'integer', n=1, size=4, endian='big')
    ret$n = readBin(f, 'integer', n=1, size=4, endian='big')
    nrow = readBin(f, 'integer', n=1, size=4, endian='big')
    ncol = readBin(f, 'integer', n=1, size=4, endian='big')
    x = readBin(f, 'integer', n=ret$n*nrow*ncol, size=1, signed=F)
    ret$x = matrix(x, ncol=nrow*ncol, byrow=T)
    close(f)
    ret
  }
  load_label_file <- function(filename) {
    f = file(filename, 'rb')
    readBin(f, 'integer', n=1, size=4, endian='big')
    n = readBin(f, 'integer', n=1, size=4, endian='big')
    y = readBin(f, 'integer', n=n, size=1, signed=F)
    close(f)
    y
  }
  path <- getwd()
  setwd(paste(path, "/mnist_db/", sep=""))
  train <- load_image_file('train-images-idx3-ubyte')
  test <- load_image_file('t10k-images-idx3-ubyte')

  train$y <- load_label_file('train-labels-idx1-ubyte')
  test$y <- load_label_file('t10k-labels-idx1-ubyte')
}

show_digit <- function(arr784, col=gray(12:1/12), ...) {
```





```
rf <- randomForest(train_x, factor(train_y), xtest=test_x,  
                    ytest=factor(test_y), ntree=numTrees)  
rf  
proc.time() - startTime
```

## **A.2 Perceptron multi-couche (PMC), carte de Kohonen (SOM), machine de Boltzmann restreinte (RBM) et réseau convolutif (CNN) sous Python**

Le script est sous format notebook. Il débute à la page suivante.

# rn\_colab\_notebook

April 14, 2020

Annexe : scripts python utilisés

Sommaire :

I. Pré-traitement des données

II. Perceptron multi-couches

III. Carte de Kohonen

IV. Machine de Boltzmann restreinte

V. Réseau convolutif

## 1 I. Pré-traitement des données

```
[1]: %tensorflow_version 1.x
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras import backend as K
from keras import activations
import numpy as np
import matplotlib.pyplot as plt
```

TensorFlow 1.x selected.

Using TensorFlow backend.

```
[ ]: img_rows, img_cols = 28, 28
num_classes = 10
```

```
[ ]: (x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
[ ]: #used to plot predictions (CNN part)
test_labels = y_test.copy()
test_images = x_test.reshape(x_test.shape[0], img_rows, img_cols)
```

```
[5]: test_images.shape
```

```
[5]: (10000, 28, 28)
```

```
[6]: # Tensorflow needs a channel in order to perform calculation
x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
x_train shape: (60000, 28, 28, 1)
```

```
60000 train samples
```

```
10000 test samples
```

```
[ ]: # convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

## 2 Perceptron multi-couche

```
[ ]: batch_size = 128
epochs = 12
```

```
[9]: %%time
model = Sequential()
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
        batch_size=batch_size,
        epochs=epochs,
        verbose=1,
        validation_data=(x_test, y_test))
```

```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

WARNING:tensorflow:From /tensorflow-1.15.2/python3.6/tensorflow\_core/python/ops/resource\_variable\_ops.py:1630: calling BaseResourceVariable.\_\_init\_\_ (from tensorflow.python.ops.resource\_variable\_ops) with constraint is deprecated and will be removed in a future version.

Instructions for updating:

If using Keras pass \*\_constraint arguments to layers.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow\_backend.py:422: The name tf.global\_variables is deprecated. Please use tf.compat.v1.global\_variables instead.

Train on 60000 samples, validate on 10000 samples

Epoch 1/12

60000/60000 [=====] - 24s 398us/step - loss: 0.4415 - accuracy: 0.8652 - val\_loss: 0.1902 - val\_accuracy: 0.9430

Epoch 2/12

60000/60000 [=====] - 22s 368us/step - loss: 0.2728 - accuracy: 0.9189 - val\_loss: 0.1400 - val\_accuracy: 0.9558

Epoch 3/12

60000/60000 [=====] - 22s 369us/step - loss: 0.2311 - accuracy: 0.9319 - val\_loss: 0.1234 - val\_accuracy: 0.9622

Epoch 4/12

60000/60000 [=====] - 22s 369us/step - loss: 0.2107 - accuracy: 0.9377 - val\_loss: 0.1190 - val\_accuracy: 0.9640

Epoch 5/12

60000/60000 [=====] - 22s 370us/step - loss: 0.1961 - accuracy: 0.9410 - val\_loss: 0.1067 - val\_accuracy: 0.9675

Epoch 6/12

60000/60000 [=====] - 22s 368us/step - loss: 0.1829 - accuracy: 0.9463 - val\_loss: 0.1050 - val\_accuracy: 0.9683

Epoch 7/12

60000/60000 [=====] - 22s 371us/step - loss: 0.1755 - accuracy: 0.9468 - val\_loss: 0.0990 - val\_accuracy: 0.9704

Epoch 8/12

60000/60000 [=====] - 22s 369us/step - loss: 0.1636 - accuracy: 0.9508 - val\_loss: 0.0894 - val\_accuracy: 0.9727

Epoch 9/12

60000/60000 [=====] - 22s 369us/step - loss: 0.1582 - accuracy: 0.9510 - val\_loss: 0.0899 - val\_accuracy: 0.9739

Epoch 10/12

60000/60000 [=====] - 22s 369us/step - loss: 0.1530 - accuracy: 0.9529 - val\_loss: 0.0919 - val\_accuracy: 0.9735

Epoch 11/12

60000/60000 [=====] - 22s 369us/step - loss: 0.1491 -

```
accuracy: 0.9544 - val_loss: 0.0917 - val_accuracy: 0.9737
Epoch 12/12
60000/60000 [=====] - 22s 369us/step - loss: 0.1447 -
accuracy: 0.9564 - val_loss: 0.0930 - val_accuracy: 0.9743
Test loss: 0.09297296016218606
Test accuracy: 0.9743000268936157
CPU times: user 2min 3s, sys: 1min 5s, total: 3min 9s
Wall time: 4min 35s

Test accuracy: 0.9743
```

### 3 Carte de Kohonen

```
[10]: pip install minisom
```

```
Requirement already satisfied: minisom in /usr/local/lib/python3.6/dist-packages
(2.2.3)
```

```
[ ]: from minisom import MiniSom
from sklearn.preprocessing import scale
import numpy as np
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train_images = x_train
```

```
[12]: x_train.shape
```

```
[12]: (60000, 28, 28)
```

```
[ ]: x_train = np.reshape(x_train, (60000,784))
```

```
[14]: x_train.shape
```

```
[14]: (60000, 784)
```

```
[ ]: x_train = scale(x_train)
```

```
[ ]: # On réduit la taille du jeu (pour plus de rapidité)
x_train_red = x_train[:2000,:]
x_train_images_red = x_train_images[:2000,:,:]
```

```
[17]: x_train_red.shape
```

```
[17]: (2000, 784)
```

```
[18]: x_train_red
```

```
[18]: array([[0., 0., 0., ..., 0., 0., 0.],
            [0., 0., 0., ..., 0., 0., 0.],
            [0., 0., 0., ..., 0., 0., 0.],
            ...,
            [0., 0., 0., ..., 0., 0., 0.],
            [0., 0., 0., ..., 0., 0., 0.],
            [0., 0., 0., ..., 0., 0., 0.]])
```

```
[ ]: #help(MiniSom)
```

```
[20]: som = MiniSom(30, 30, 784, sigma=4,
                    learning_rate=0.5, neighborhood_function='triangle')
som.pca_weights_init(x_train_red)
print("Training...")
som.train_random(x_train_red, 5000, verbose=True) # random training
print("\n...ready!")
```

```
/usr/local/lib/python3.6/dist-packages/minisom.py:304: ComplexWarning: Casting
complex values to real discards the imaginary part
```

```
self._weights[i, j] = c1*pc[pc_order[0]] + c2*pc[pc_order[1]]
```

```
Training...
```

```
[ 5000 / 5000 ] 100% - 0:00:00 left
```

```
quantization error: 0.2963368694831614
```

```
topographic error: 0.1765
```

```
...ready!
```

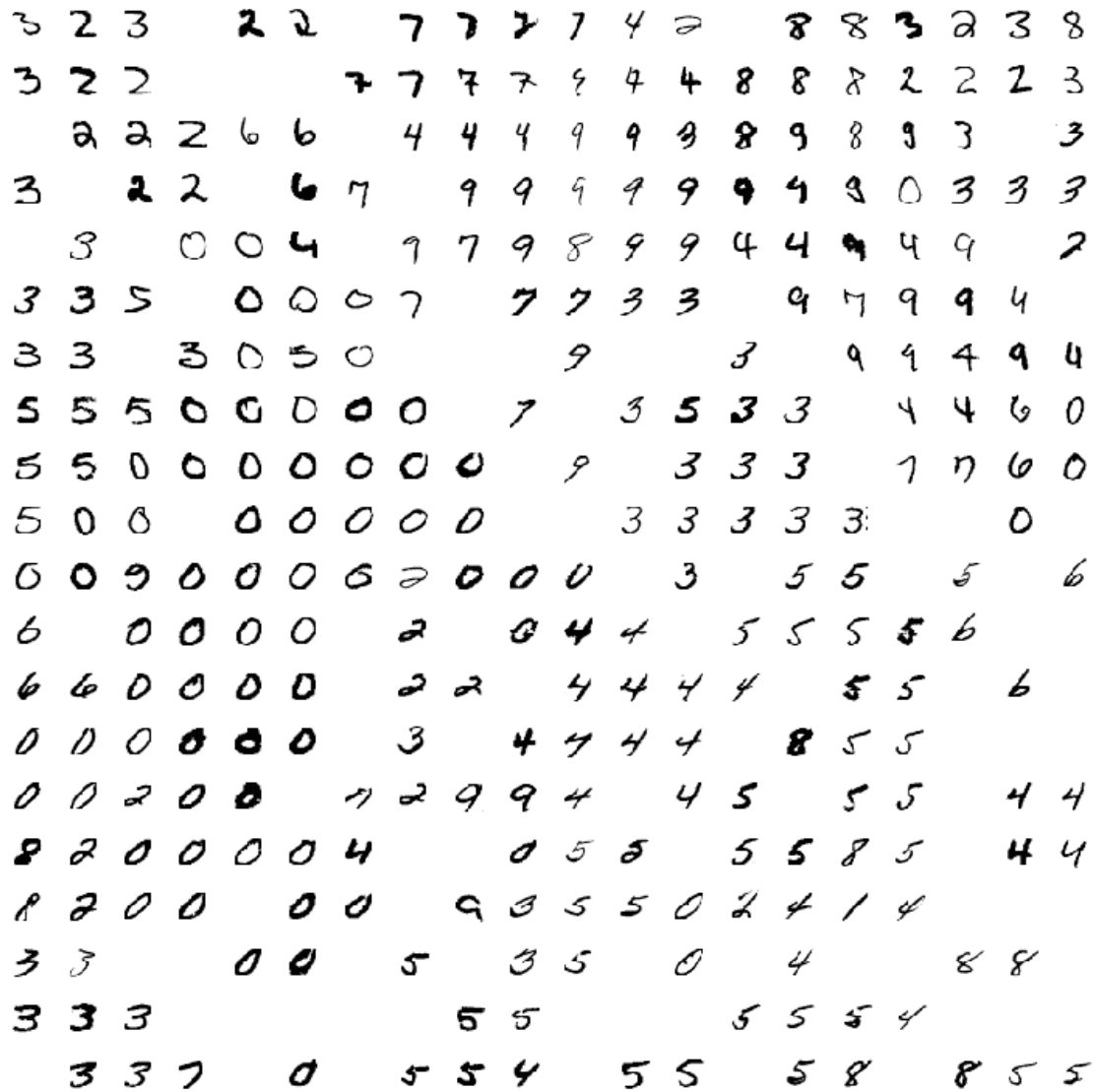
```
[21]: import matplotlib.pyplot as plt

num = y_train[:2000] # num[i] is the digit represented by x_train[i]
plt.figure(figsize=(8, 8))
wmap = {}
im = 0
for x, t in zip(x_train_red, num): # scatterplot
    w = som.winner(x)
    wmap[w] = im
    #plt.text(w[0]+.5, w[1]+.5, str(t),
    #         color=plt.cm.rainbow(t / 10.), fontdict={'weight': 'bold',
    #         ↪ 'size': 11})
    im = im + 1
#plt.axis([0, som.get_weights().shape[0], 0, som.get_weights().shape[1]])
#plt.show()
```

```
<Figure size 576x576 with 0 Axes>
```

```
[22]: plt.figure(figsize=(10, 10), facecolor='white')
cnt = 0
for j in reversed(range(20)): # images mosaic
    for i in range(20):
        plt.subplot(20, 20, cnt+1, frameon=False, xticks=[], yticks=[])
        if (i, j) in wmap:
            plt.imshow(x_train_images_red[wmap[(i, j)]],
                       cmap='Greys', interpolation='nearest')
        else:
            plt.imshow(np.zeros((28, 28)), cmap='Greys')
        cnt = cnt + 1

plt.tight_layout()
plt.show()
```



The image displays a 20x20 grid of handwritten digits, likely from the MNIST dataset. The digits are arranged in a mosaic pattern, with some digits appearing more frequently than others. The grid shows a variety of handwritten styles and orientations. The digits are displayed in grayscale, and the background is white. The digits are arranged in a grid that is 20 rows by 20 columns. The digits are arranged in a mosaic pattern, with some digits appearing more frequently than others. The grid shows a variety of handwritten styles and orientations. The digits are displayed in grayscale, and the background is white. The digits are arranged in a grid that is 20 rows by 20 columns.

```
[23]: import numpy as np
vec = np.array([[1,2,3],[6,5,4]])
vec
```

```
[23]: array([[1, 2, 3],
          [6, 5, 4]])
```

```
[24]: vec = np.reshape(vec, 6)
vec
```

```
[24]: array([1, 2, 3, 6, 5, 4])
```

```
[ ]: from sklearn import datasets
from sklearn.preprocessing import scale

# load the digits dataset from scikit-learn
digits = datasets.load_digits(n_class=10)
data = digits.data # matrix where each row is a vector that represent a digit.
data = scale(data)
```

```
[26]: data
```

```
[26]: array([[ 0.          , -0.33501649, -0.04308102, ..., -1.14664746,
          -0.5056698 , -0.19600752],
          [ 0.          , -0.33501649, -1.09493684, ...,  0.54856067,
          -0.5056698 , -0.19600752],
          [ 0.          , -0.33501649, -1.09493684, ...,  1.56568555,
           1.6951369 , -0.19600752],
          ...,
          [ 0.          , -0.33501649, -0.88456568, ..., -0.12952258,
          -0.5056698 , -0.19600752],
          [ 0.          , -0.33501649, -0.67419451, ...,  0.8876023 ,
          -0.5056698 , -0.19600752],
          [ 0.          , -0.33501649,  1.00877481, ...,  0.8876023 ,
          -0.26113572, -0.19600752]])
```

```
[27]: som = MiniSom(30, 30, 64, sigma=4,
                    learning_rate=0.5, neighborhood_function='triangle')
som.pca_weights_init(data)
print("Training...")
som.train_random(data, 5000, verbose=True) # random training
print("\n...ready!")
```

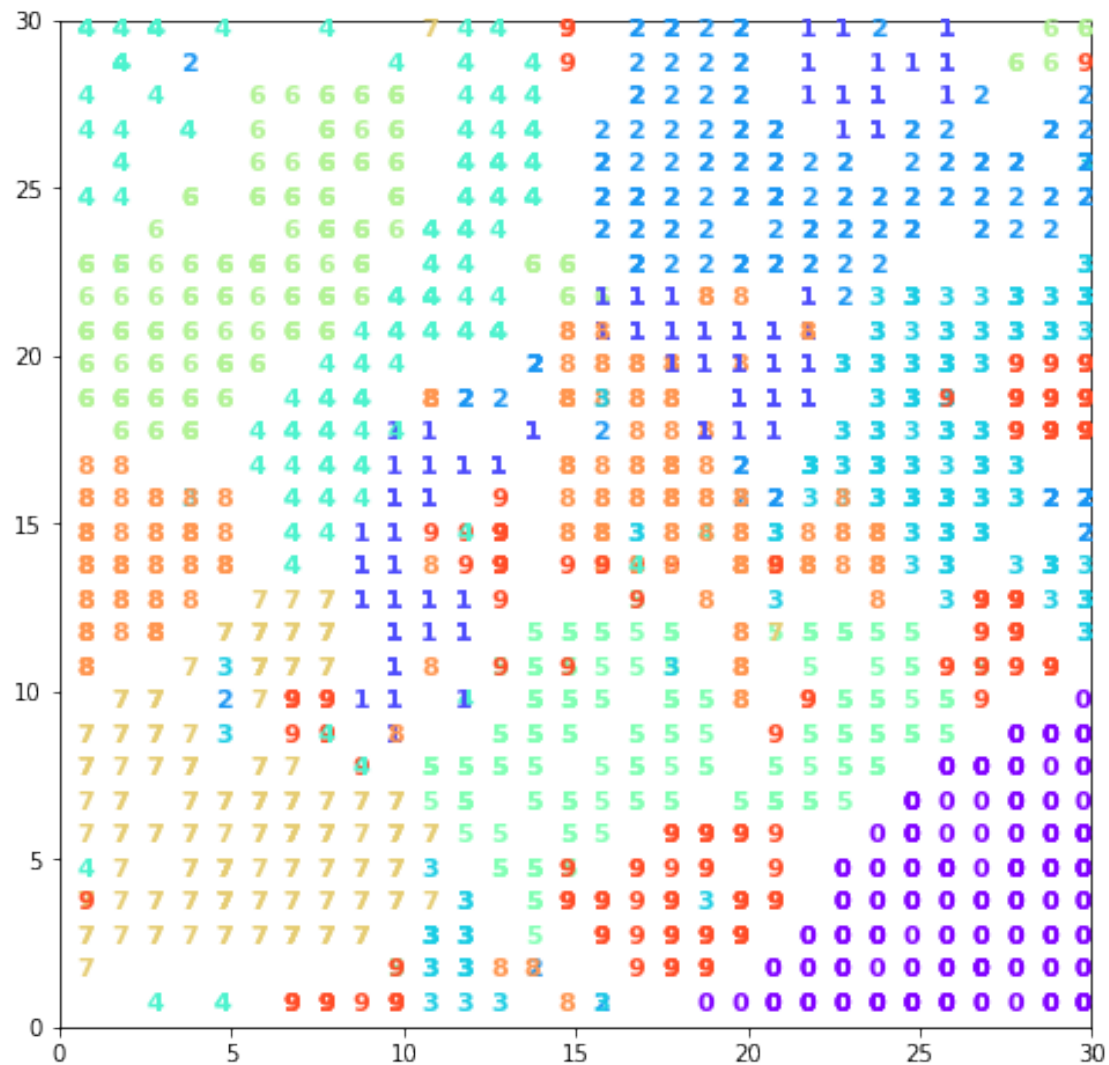
```
Training...
[ 5000 / 5000 ] 100% - 0:00:00 left
quantization error: 0.1706860439276576
```



topographic error: 0.15080690038953812

...ready!

```
[28]: import matplotlib.pyplot as plt
num = digits.target # num[i] is the digit represented by data[i]
plt.figure(figsize=(8, 8))
wmap = {}
im = 0
for x, t in zip(data, num): # scatterplot
    w = som.winner(x)
    wmap[w] = im
    plt.text(w[0]+.5, w[1]+.5, str(t),
             color=plt.cm.rainbow(t / 10.), fontdict={'weight': 'bold', 'size': 11})
    im = im + 1
plt.axis([0, som.get_weights().shape[0], 0, som.get_weights().shape[1]])
plt.show()
```



## 4 Machine de Boltzmann Restreinte

Application au jeu MNIST

```
[ ]: from sklearn.model_selection import train_test_split
from sklearn.preprocessing import scale
from sklearn.metrics import classification_report
from sklearn.neural_network import BernoulliRBM
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
import numpy as np
import argparse
import time
```

```
import cv2
```

```
[30]: (x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train = np.reshape(x_train, (60000,784))
x_test = np.reshape(x_test, (10000,784))
x_test = scale(x_test)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/preprocessing/_data.py:173:
UserWarning: Numerical issues were encountered when centering the data and might
not be solved. Dataset may contain too large values. You may need to prescale
your features.
```

```
warnings.warn("Numerical issues were encountered "
/usr/local/lib/python3.6/dist-packages/sklearn/preprocessing/_data.py:190:
UserWarning: Numerical issues were encountered when scaling the data and might
not be solved. The standard deviation of the data is probably very close to 0.
warnings.warn("Numerical issues were encountered "
```

```
[ ]: rbm = BernoulliRBM(verbose = True, random_state=0)
```

```
[ ]: from sklearn.linear_model import LogisticRegression
logistic = LogisticRegression(solver='newton-cg', tol=1)
```

```
[ ]: rbm.learning_rate = 0.06
rbm.n_iter = 10
rbm.n_components = 100
logistic.C = 6000

rbm_features_classifier = Pipeline(steps=[("rbm", rbm), ("logistic", logistic)])
```

```
[36]: rbm_features_classifier.fit(x_train, y_train)
print(classification_report(y_test, rbm_features_classifier.predict(x_test)))
```

```
[BernoulliRBM] Iteration 1, pseudo-likelihood = -464090086.39, time = 15.54s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -928180774.31, time = 17.11s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -1392271438.93, time = 17.05s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -1856362123.75, time = 17.24s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -2320452798.95, time = 17.19s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -2784543461.55, time = 17.12s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -3248634144.47, time = 17.10s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -3712724814.04, time = 17.06s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -4176815491.38, time = 17.13s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -4640906193.15, time = 17.21s
precision    recall  f1-score   support
```

```
0           0.00      0.00      0.00      980
```

1	0.11	1.00	0.20	1135
2	0.00	0.00	0.00	1032
3	0.00	0.00	0.00	1010
4	0.00	0.00	0.00	982
5	0.00	0.00	0.00	892
6	0.00	0.00	0.00	958
7	0.00	0.00	0.00	1028
8	0.00	0.00	0.00	974
9	0.00	0.00	0.00	1009
accuracy			0.11	10000
macro avg	0.01	0.10	0.02	10000
weighted avg	0.01	0.11	0.02	10000

```
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py:1272:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
```

```
_warn_prf(average, modifier, msg_start, len(result))
```

Commentaire : On obtient une justesse de 11% ce qui est légèrement supérieur à un classifieur aléatoire (10%).

## 5 Réseau convolutionnel

Application au jeu MNIST

On se rappelle que l'on a obtenu le score de 97.35% de classification correcte avec un perceptron multi-couche. Voyons ce quel score nous pouvons obtenir avec un réseau de neurones convolutionnel.

```
[ ]: from keras.layers import Conv2D, MaxPooling2D
```

```
batch_size = 128
epochs = 12
```

```
[ ]: img_rows, img_cols = 28, 28
num_classes = 10
```

```
[ ]: (x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
[ ]: #used to plot predictions (CNN part)
test_labels = y_test.copy()
test_images = x_test.reshape(x_test.shape[0], img_rows, img_cols)
```

```
[44]: test_images.shape
```

```
[44]: (10000, 28, 28)
```

```
[45]: # Tensorflow needs a channel in order to perform calculation
x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

```
[ ]: # convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

Construction du réseau.

```
[47]: model = Sequential()
model.add(Conv2D(filters=32,
                 kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape,
                 padding='valid',
                 strides=(1,1)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2),
                      strides=None,
                      padding='valid'))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
        batch_size=batch_size,
        epochs=epochs,
        verbose=1,
```

```
        validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Train on 60000 samples, validate on 10000 samples

Epoch 1/12

60000/60000 [=====] - 13s 219us/step - loss: 0.2736 - accuracy: 0.9158 - val\_loss: 0.0580 - val\_accuracy: 0.9813

Epoch 2/12

60000/60000 [=====] - 9s 143us/step - loss: 0.0892 - accuracy: 0.9742 - val\_loss: 0.0457 - val\_accuracy: 0.9841

Epoch 3/12

60000/60000 [=====] - 9s 144us/step - loss: 0.0666 - accuracy: 0.9803 - val\_loss: 0.0356 - val\_accuracy: 0.9877

Epoch 4/12

60000/60000 [=====] - 9s 144us/step - loss: 0.0547 - accuracy: 0.9833 - val\_loss: 0.0348 - val\_accuracy: 0.9883

Epoch 5/12

60000/60000 [=====] - 9s 144us/step - loss: 0.0469 - accuracy: 0.9857 - val\_loss: 0.0309 - val\_accuracy: 0.9898

Epoch 6/12

60000/60000 [=====] - 9s 143us/step - loss: 0.0420 - accuracy: 0.9877 - val\_loss: 0.0286 - val\_accuracy: 0.9901

Epoch 7/12

60000/60000 [=====] - 9s 144us/step - loss: 0.0366 - accuracy: 0.9886 - val\_loss: 0.0279 - val\_accuracy: 0.9908

Epoch 8/12

60000/60000 [=====] - 9s 144us/step - loss: 0.0352 - accuracy: 0.9894 - val\_loss: 0.0263 - val\_accuracy: 0.9910

Epoch 9/12

60000/60000 [=====] - 9s 144us/step - loss: 0.0309 - accuracy: 0.9905 - val\_loss: 0.0268 - val\_accuracy: 0.9920

Epoch 10/12

60000/60000 [=====] - 9s 145us/step - loss: 0.0289 - accuracy: 0.9911 - val\_loss: 0.0275 - val\_accuracy: 0.9915

Epoch 11/12

60000/60000 [=====] - 9s 145us/step - loss: 0.0264 - accuracy: 0.9916 - val\_loss: 0.0263 - val\_accuracy: 0.9921

Epoch 12/12

60000/60000 [=====] - 9s 147us/step - loss: 0.0262 - accuracy: 0.9919 - val\_loss: 0.0249 - val\_accuracy: 0.9917

Test loss: 0.02490975712309846

Test accuracy: 0.9916999936103821

On obtient une précision accrue : 99% de bonnes réponses. Regardons de plus près la qualité des prédictions que notre réseau convolutionnel nous donne sur le jeu de généralisation.

```
[ ]: predictions = model.predict(x_test)
```

Pour chaque photo d'un chiffre manuscrit, on obtient un vecteur de taille 10 dont chacune des composantes représente la probabilité estimée par notre algorithme d'avoir le chiffre de la composante. Si la deuxième composante vaut 0.5, notre méthode estime qu'il y a une chance sur deux que le chiffre en question soit un 1.

```
[49]: predictions[0]
```

```
[49]: array([4.3428663e-11, 1.4906426e-09, 1.4604564e-09, 3.4959186e-08,  
        2.4577734e-11, 1.2750767e-10, 7.9236761e-14, 1.0000000e+00,  
        1.4820879e-11, 1.8244178e-08], dtype=float32)
```

La somme des 10 probabilités ci-dessus nous donne 1. Nous pouvons regarder nos images et y associer les prédictions faites.

```
[50]: preds = np.array([np.argmax(pr) for pr in predictions])  
print(preds)  
#print(test_labels)  
#print(preds[preds!=test_labels])  
print("Nombre de photos mal classes = {}".format(sum(preds!=test_labels)))  
#print(np.argwhere(preds!=test_labels))
```

```
[7 2 1 ... 4 5 6]
```

Nombre de photos mal classes = 83

```
[ ]: idx_wrong = np.argwhere(preds!=test_labels)  
idx_wrong = idx_wrong.flatten()  
#for i in idx_wrong[:10]:  
#    print(i)
```

```
[ ]: # block written by François Chollet  
def plot_image(i, predictions_array, true_label, img):  
    predictions_array, true_label, img = predictions_array, true_label[i], img[i]  
    plt.grid(False)  
    plt.xticks([])  
    plt.yticks([])  
  
    plt.imshow(img, cmap=plt.cm.binary)  
  
    predicted_label = np.argmax(predictions_array)  
  
    if predicted_label == true_label:  
        color = 'blue'  
    else:  
        color = 'red'  
  
    plt.xlabel("{} {:.2f}% ({})" .format(predicted_label,
```

```

100*np.max(predictions_array),
true_label),
color=color)

def plot_value_array(i, predictions_array, true_label):
    predictions_array, true_label = predictions_array, true_label[i]
    plt.grid(False)
    plt.xticks(range(10))
    plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array, color="#777777")
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array)

    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('blue')

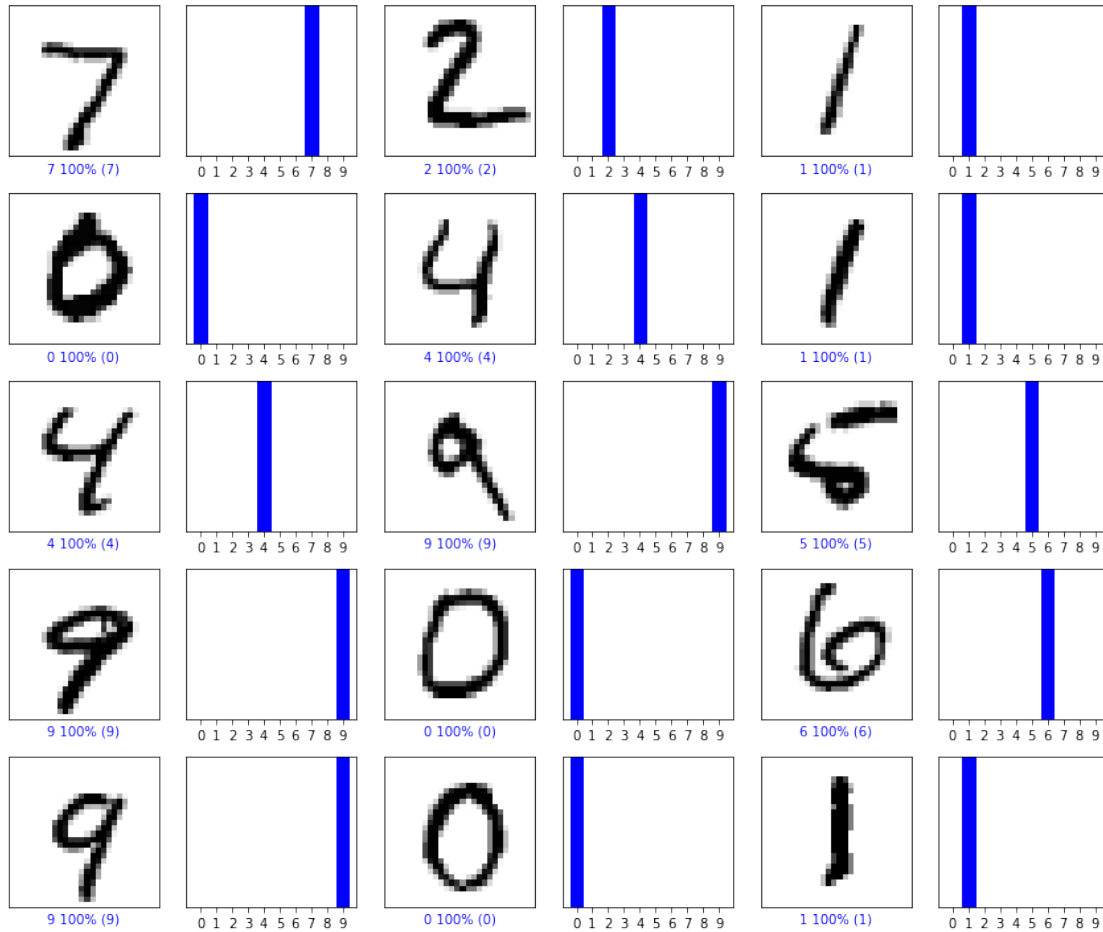
```

```

[53]: num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    plot_image(i, predictions[i], test_labels, test_images)
    plt.subplot(num_rows, 2*num_cols, 2*i+2)
    plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()

```



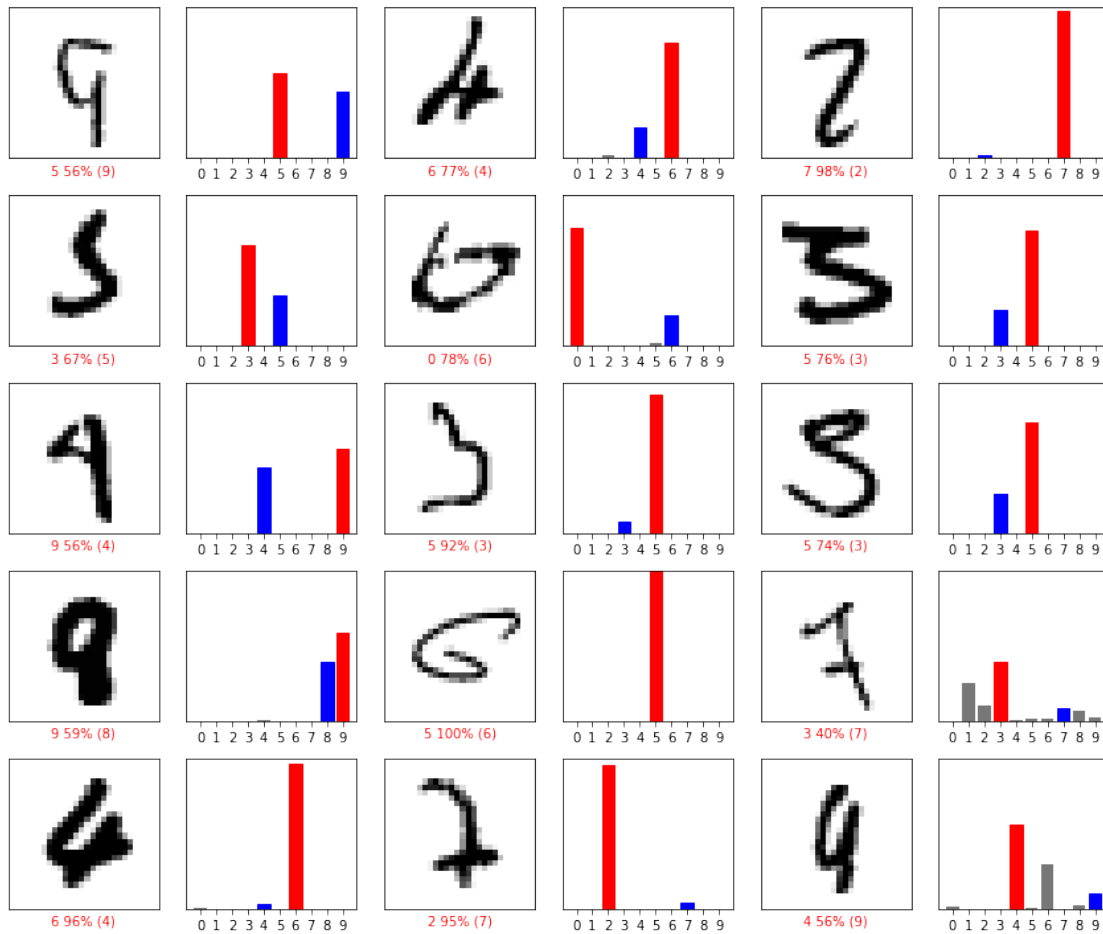


Observation : Ici nous avons pris les 15 premières observation du jeu de généralisation. Celui-ci en contient 10 000, étant donné notre justesse (99 %), il y a peu de chance d'obtenir une mauvaise classification parmi ces 15 observations.

Regardons les exemples qui ont mal été classés par notre algorithme. Cela nous permettrait de savoir si un humain aurait été meilleur.

```
[54]: num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
cpt = 0
for i in idx_wrong[:num_images]:
    plt.subplot(num_rows, 2*num_cols, 2*cpt+1)
    plot_image(i, predictions[i], test_labels, test_images)
    plt.subplot(num_rows, 2*num_cols, 2*cpt+2)
    plot_value_array(i, predictions[i], test_labels)
    cpt+=1
plt.tight_layout()
```

```
plt.show()
```



Observations : Souvent, la classification effectuée par l'oeil humain aurait été meilleure que celle de notre algorithme. Il suffit de regarder les deux dernières lignes pour s'en persuader. Par ailleurs, on observe sur ces 15 exemples que la deuxième probabilité la plus élevée représente le bon chiffre.

[ ]:

## References

- [Cybenko, 1989] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314.
- [Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. <http://www.deeplearningbook.org>.
- [Goodfellow et al., 2013] Goodfellow, I., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013). Maxout networks. *30th International Conference on Machine Learning, ICML 2013*, 1302.
- [Haykin, 2008] Haykin, S. (2008). *Neural Networks and Learning Machines*, page 429. Pearson.
- [Hinton, 2002] Hinton, G. (2002). Training products of experts by minimizing contrastive divergence. *Neural computation*, 14:1771–800.
- [Hinton, 2010] Hinton, G. (2010). A practical guide to training restricted boltzmann machines. *Momentum*, 9:926–947.
- [Ho, 1995] Ho, T. K. (1995). Random decision forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, ICDAR ’95, page 278, USA. IEEE Computer Society.
- [Hornik et al., 1989] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feed-forward networks are universal approximators. *Neural Networks*, 2(5):359 – 366.
- [Kohonen, 1982] Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69.
- [Kohonen, 2001] Kohonen, T. (2001). *Self-organizing maps*. Number 30 in Springer series in information sciences. Springer, Berlin ; New York, 3rd ed edition.
- [Le Roux and Bengio, 2008] Le Roux, N. and Bengio, Y. (2008). Representational power of restricted boltzmann machines and deep belief networks. *Neural computation*, 20:1631–1649.
- [Long and Servedio, 2010] Long, P. and Servedio, R. (2010). Restricted boltzmann machines are hard to approximately evaluate or simulate. *ICML 2010 - Proceedings, 27th International Conference on Machine Learning*, pages 703–710.
- [Smolensky, 1986] Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. In *Parallel distributed processing: Explorations in the microstructure of cognition*, pages 194–281-. MIT Press, Cambridge, MA.
- [Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

- [Sutskever and Tieleman, 2010] Sutskever, I. and Tieleman, T. (2010). On the convergence properties of contrastive divergence. *Journal of Machine Learning Research - Proceedings Track*, 9:789–795.
- [Tibshirani, 1996] Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58:267–288.