~~Valentyn Geruchta — 3703995~~
~~Léane Salais — 16003618~~
Rapport n°2 – 27 novembre

rdfia

réseaux de neurones

Dans les TMEs 3 et 4, on propose une tâche de classification simple sous PyTorch.
Comme on le verra en comparant avec des SVM, utiliser des réseaux de neurones, du Deep Learning, permet de repérer des frontières de classification plus complexes en apprenant des fonctions non linéaires. La difficulté réside cependant dans la modulation des hyperparamètres (architecture, pas d'apprentissage…) qui rend l'implémentation imprévisible pour un utilisateur inexpérimenté ; on prend donc le temps de maîtriser la théorie avec quelques questions de compréhension.

Jeu de données

La différenciation des données présentées au réseau en *train* et en *test* est cruciale pour la validité de l'optimisation. La base de données doit être partagée en suivant certaines règles.

1 A quoi servent les ensembles d'apprentissage, de validation et de test ?

L'objectif d'un modèle de Machine Learning est d'extraire les dynamiques inhérentes à des données. Pour ce faire, **on les lui fournit dans un ensemble d'apprentissage (ou *train*) à étudier**.

Cependant, il n'est pas raisonnable de l'évaluer sur ce même ensemble. Il en apprendrait « par cœur » les spécificités jusqu'à obtenir un score parfait et ne saurait pas exporter sa modélisation à d'autres données : c'est le surapprentissage, qu'il faut éviter à tout prix. **Pour s'assurer que le modèle saura généraliser et analyser cette généralisation sans biais, on crée donc séparément un ensemble d'évaluation (ou *test*)**.

En Deep Learning, **un dernier ensemble dit de validation (ou *dev*) sert à optimiser les hyperparamètres** sans tricher en les ajustant exactement à l'ensemble de test. Même s'il est issu de la même distribution, leur séparation est nécessaire pour que la validité de l'optimisation ne soit pas limitée à un ensemble précis.

Après l'entraînement, on recherche donc le meilleur score possible en *dev*, puis on exporte les paramètres optimaux trouvés pour obtenir le meilleur score possible en *test*.

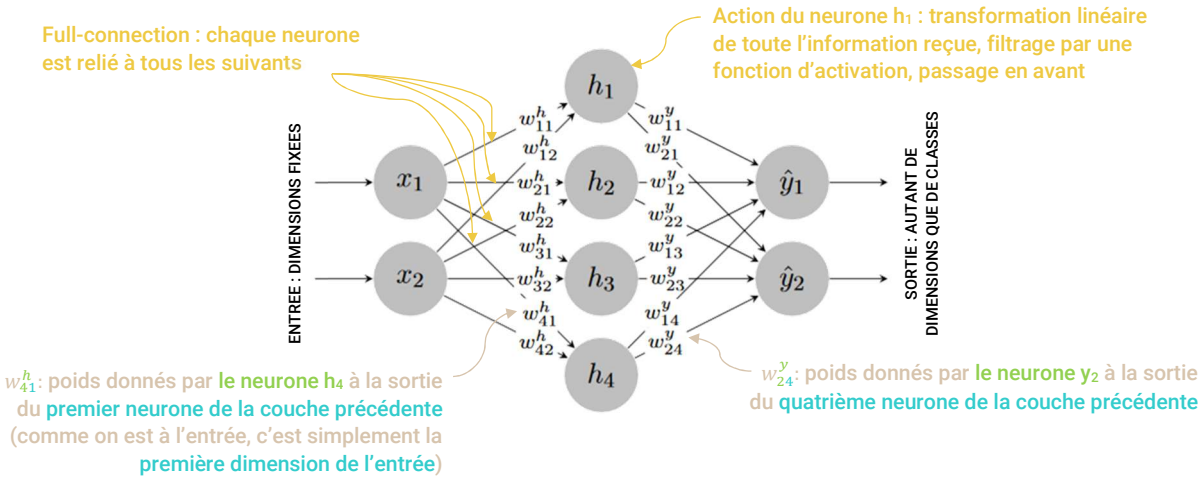
2 Quelle est l'influence du nombre N d'exemples ?

Avoir plus d'exemples sert à diversifier les cas rencontrés. Exposer le modèle à des situations différentes le met en difficulté et l'oblige à améliorer sa généralisation (ce qui donne de meilleur résultats en *test*). A l'inverse, un modèle qui n'a connu qu'une sous-partie des possibilités ne ferait que du surapprentissage sur le *train*. **Donc, même si l'entraînement sur plus d'exemples est plus long, plus difficile pour chaque passe à travers les données, il évite le surapprentissage et produit à terme un meilleur score en test.**

Architecture du réseau

Organisé en plusieurs couches, un réseau consiste en un ensemble d'unités de calcul (les neurones) réparties entre elles.
Dans le cas basique comme ici, l'information donnée en entrée parcourt les couches de gauche à droite jusqu'à la sortie. Les neurones de chaque couche sont tous connectés à ceux de la suivante, qui réceptionnent leur *output* et agissent à partir de lui ; on parle de *fully-connected (FC) layers*.

Chaque neurone fait subir une transformation linéaire à l'information qu'il a reçue de ses collègues. Il la multiplie par un vecteur de poids W qui lui est propre (ce qui lui permet de pondérer selon sa décision l'output de chacun des neurones précédents) et lui ajoute un biais b.
Le résultat subit ensuite une activation (transformation non linéaire) sur le principe des seuils d'excitation cérébraux : le passage de l'information transformée par un neurone au suivant dépend donc de l'intensité de la stimulation.



3 Pourquoi est-il important d'ajouter des fonctions d'activation entre des transformations linéaires ?

Les fonctions d'activation complexifient le modèle. A chaque couche, elles amplifient la sortie de certains neurones, en silencient d'autres, les font donc compter à divers degrés dans le résultat selon la force de leur signal : **grâce à elles, les composantes de chaque couche ne sont plus toutes considérées *exactement de la même manière*, et le réseau devient capable d'apprendre des fonctions non linéaires**.
A l'inverse, une transformation linéaire s'applique uniformément à un ensemble. Sans activation, quel que soit le nombre de couches linéaires successives, on obtiendrait donc à terme le même résultat qu'avec une seule couche optimisée. Le modèle serait incapable d'apprendre des frontières de décision complexes.

4 Quelles sont les tailles n_x, n_h, n_y sur la figure ? En pratique, comment ces tailles sont-elles choisies ?

→ **n_x (ici 2) correspond à la dimension d'entrée**. n_x n'est pas choisie, elle dépend de l'aspect des données disponibles (note : l'apprentissage est facilité quand elle est petite, d'où l'importance du *preprocessing*).
→ **n_y (ici 2) est le nombre de neurones de la couche de sortie,** celle qui convertit les calculs internes au réseau en prédiction finale. Dans un problème de classification, il doit être égal au nombre de classes possibles, il n'est donc pas choisi non plus et dépend du *setup* du problème.
→ **n_h (ici 4) est la dimension latente**, le nombre de neurones dans la couche cachée (une seule ici, bien plus habituellement). La dimension de chaque couche du réseau est un hyperparamètre à optimiser, **c'est un choix qu'on fait souvent via GridSearch de sorte à maximiser le score obtenu sur l'ensemble de *dev***.

Transformées par plusieurs couches, les données d'entrée x finissent par atteindre la couche de sortie. Ses neurones intègrent ce dernier input et chacun génère une *valeur* $\tilde{y}$ associée à une classe. Reste à convertir $\tilde{y}$ en prédiction probabiliste $\hat{y}$ (via un SoftMax) et à la comparer avec une vérité y.

5 Que représentent les vecteurs y et $\hat{y}$? Quelle est la différence entre ces deux quantités ?

y représente la vérité, la cible d'apprentissage. C'est vers elle que la prédiction du modèle, $\hat{y}$, doit tendre.
En classification mono-label, y est encodée en one-hot (la classe réelle est valuée à 1 et les autres à zéro). Alors que y est connue, $\hat{y}$ évolue ; au cours de l'apprentissage, on fait en sorte que la différence entre les deux diminue autant que possible. Cette différence s'appelle le *coût* et son gradient servira à mettre à jour les paramètres à chaque itération jusqu'à la convergence (cf. sections suivantes).

6 **Pourquoi utiliser une fonction SoftMax en sortie ?**

La fonction SoftMax comme dernière transformation est adaptée à la classification. Elle renvoie ce qu'on peut interpréter comme une distribution de probabilité sur les classes ; pour chaque exemple à classer, elle donne un vecteur de norme 1 avec autant de valeurs que de classes – toutes positives.
Une conversion en SoftMax est aussi plus informative sur l'hésitation du modèle qu'un one-hot encoding (comme pour y) pour la classe la plus vraisemblable. Le modèle se trompe souvent en début d'apprentissage ; encoder sa prédiction ainsi permet de comprendre ce qui lui semble *plus ou moins plausible* pour chaque exemple à une itération donnée. On pourra alors le pénaliser de façon plus ciblée.
Sa pente douce rend aussi SoftMax dérivable, donc utilisable en backpropagation (cf. section liée).

7 **Ecrire les équations permettant d'effectuer la passe forward du réseau de neurones, c'est-à-dire permettant de produire successivement $\tilde{h}$, h, $\tilde{y}$ et $\hat{y}$ à partir de x.**

On a commenté p. 2 l'activité d'un neurone, on étudie maintenant la sortie par couche : les vecteurs calculés par chacun de ses neurones sont réunis dans une matrice, et les transformations se font en parallèle (multiplication par une *matrice* de poids, somme avec un *vecteur* de biais...). Cette parallélisation permet d'accélérer les calculs en utilisant la vectorisation sous Python, c'est donc ce qui se fait en pratique.

PREMIERE TRANSFORMATION LINEAIRE : $\tilde{h} = W_h x + b_h$. La sortie $\tilde{h}$ est un vecteur de taille n_h.

W_h est une ***matrice*** de poids de taille $n_h \times n_x$
b_h est un ***vecteur*** de biais de taille n_h
x est un vecteur de features de taille n_x

$$\begin{bmatrix} \tilde{h}_1 \\ \vdots \\ \tilde{h}_{n_h} \end{bmatrix} = \begin{bmatrix} W_{h,11} & \cdots & W_{h,1n_x} \\ \vdots & \ddots & \vdots \\ W_{h,\,n_h 1} & \cdots & W_{h,n_h n_x} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{n_x} \end{bmatrix} + \begin{bmatrix} b_1^h \\ \vdots \\ b_{n_h}^h \end{bmatrix}$$

$$\boldsymbol{\tilde{h}_i = \sum_{j=1}^{n_x} W_{h,ij} \cdot x_j + b_i^h}$$

ACTIVATION NON-LINEAIRE : $h = \tanh(\tilde{h})$. La sortie h est de même dimension que $\tilde{h}$.

$\tilde{h}$ est un vecteur de taille n_h

$$\begin{bmatrix} h_1 \\ \vdots \\ h_{n_h} \end{bmatrix} = \tanh \begin{bmatrix} \tilde{h}_1 \\ \vdots \\ \tilde{h}_{n_h} \end{bmatrix} \quad \text{et} \quad \boldsymbol{h_i = \tanh(\tilde{h}_i)}$$

SECONDE TRANSFORMATION LINEAIRE : $\tilde{y} = W_y h + b_y$. La sortie $\tilde{y}$ est un vecteur de taille n_y.

W_y est une matrice de poids de taille $n_y \times n_h$
b_y est un vecteur de biais de taille n_y
h est un vecteur de taille n_h

$$\begin{bmatrix} \tilde{y}_1 \\ \vdots \\ \tilde{y}_{n_y} \end{bmatrix} = \begin{bmatrix} W_{y,11} & \cdots & W_{y,1n_h} \\ \vdots & \ddots & \vdots \\ W_{y,\,n_y 1} & \cdots & W_{y,n_y n_h} \end{bmatrix} \begin{bmatrix} h_1 \\ \vdots \\ h_{n_h} \end{bmatrix} + \begin{bmatrix} b_1^y \\ \vdots \\ b_{n_y}^y \end{bmatrix}$$

$$\boldsymbol{\tilde{y}_i = \sum_{j=1}^{n_h} W_{y,ij} \cdot h_j + b_i^y}$$

ACTIVATION NON-LINEAIRE : $\hat{y} = SoftMax(\tilde{y})$. La sortie $\hat{y}$ est de même dimension que $\tilde{y}$.

$\tilde{y}$ est un vecteur de taille n_y

$$\begin{bmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_{n_y} \end{bmatrix} = \frac{\exp \begin{bmatrix} \tilde{y}_1 \\ \vdots \\ \tilde{y}_{n_y} \end{bmatrix}}{\sum_{i=1}^{n_y} \exp(\tilde{y}_i)} \quad \text{et} \quad \boldsymbol{\hat{y}_i = \frac{e^{\tilde{y}_i}}{\sum \tilde{y}_i}}$$

Fonction de coût

On a besoin de mesurer la différence entre la sortie du modèle et la vérité terrain pour la minimiser. Les modes de mesure, les fonctions de coût, s'adaptent différemment aux problèmes concernés.

8 **Pour l'entropie croisée et l'erreur quadratique, comment les $\hat{y}$ doivent-ils varier pour diminuer la loss ?**

Minimiser (diminuer) une fonction, c'est annuler sa dérivée. Or, l'objectif de l'apprentissage du réseau est d'avoir une prédiction $\hat{y}$ très proche de y : la fonction de coût doit donc atteindre un minimum quand $\hat{y} \sim y$.

C'est bien le cas pour l'erreur quadratique…

$l(y, \hat{y}) = \sum_i \|y_i - \hat{y}_i\|^2$ et $\frac{\partial l}{\partial \hat{y}} = 2(y - \hat{y})$

La dérivée s'annule quand la différence tend vers 0, donc le minimum est atteint quand $\hat{y} \sim y$.

…et l'entropie croisée.

$l(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$ et $\frac{\partial l}{\partial \hat{y}} = -\frac{y}{\hat{y}}$

La dérivée s'annule quand le quotient tend vers 1, donc le minimum est atteint quand $\hat{y} \sim y$.

9 En quoi ces fonctions sont-elles plus adaptées aux problèmes de classification ou de régression ?

→ **L'erreur quadratique est adaptée aux problèmes de régression**. Son gradient est *proportionnel* à $y - \hat{y}$ et il est très simple à calculer : ce n'est pas justifiable d'utiliser autre chose. Mais cette méthode n'est pas exportable à la classification, où on ne regarde pas la valeur absolue des vraisemblances de chaque classe (pas une distribution normale), mais de *combien* celle de la classe réelle domine. On ne devrait pas donner de poids aux valeurs exactes du reste.
→ **Le coût entropie croisée est donc plus adapté à la classification** dans la mesure où il sert à comparer des probabilités : il prend en compte le fait que les valeurs somment à 1 (distribution multinomiale) et permet d'encourager la dominance d'une classe sur les autres.

Méthode d'apprentissage et backpropagation

Après la passe en avant des données (*forward*), le gradient du coût doit être répercuté sur les paramètres de chaque neurone, poids et biais. Cela se fait via un mécanisme de rétropropagation dont on développera ici les équations. On répètera cette boucle pour un certain nombre d'itérations. En plus des hyperparamètres (dont le learning rate η), la façon d'étudier les données de *train* – toutes à la fois, par élément ou par sous-ensemble – influe sur la portée du calcul du gradient du coût, et donc sur la rapidité de la convergence vers le paramétrage optimal des couches.

10 Quels sont les avantages et inconvénients des variantes de descente de gradient, entre batch, stochastique mini-batch et stochastique online ? Laquelle semble la plus raisonnable à utiliser ?

→ La descente *batch* attend d'observer la totalité des données pour mettre à jour les paramètres : il y a donc une mise à jour par passe sur les données (*epoch*). On se fait ainsi une très bonne idée du contenu de la base, mais l'inconvénient est que l'apprentissage est lent si elle est grande.

→ La descente stochastique *online* est adaptée lorsqu'on souhaite avoir une évolution immédiate. La mise à jour des paramètres dépend d'un seul exemple tiré au hasard : cela fait toujours une mise à jour par *epoch*, donc le nombre d'itérations à convergence est à peu près le même ; mais on gagne en temps de calcul puisqu'on ignore à chaque fois la majorité des données. L'apprentissage est cependant chaotique avec des paramètres instables d'une itération à l'autre selon le degré de représentativité des tirages sucessifs.

→ La descente stochastique mini-batch est la plus utilisée en pratique, car elle combine les avantages des autres. Les paramètres sont mis à jour après l'étude d'une sous-partie des données, un mini-batch : il y a donc plusieurs mises à jour par *epoch*, le temps d'observer tous les mini-batchs. Cette technique allie une plus grande vitesse d'apprentissage qu'en batch (le temps entre les mises à jour est plus court) avec plus de stabilité qu'en stochastique online (une mise à jour utilise plus de données et est donc plus informative).

11 Quelle est l'influence du learning rate η sur l'apprentissage ?

η, aussi appelé pas de gradient, influe sur la convergence. S'il est trop petit, elle sera très lente, car on s'approche très peu de l'optimum à chaque itération ; mais ce peut être un choix de sécurité. De fait, s'il est trop grand, on peut rater le minimum global ou se retrouver dans une situation de divergence totale. Une bonne pratique est de faire décroître η au fil des itérations pour atteindre la région cible le plus vite possible au départ mais augmenter la précision d'apprentissage en s'approchant du paramétrage optimal.

La répercussion du gradient du coût sur les paramètres est optimisée grâce à l'algorithme *backprop*. C'est grâce à lui que l'apprentissage se finit dans un temps acceptable :

12 Comparer la complexité (en fonction du nombre de couches du réseau) du calcul des gradients de la *loss* par rapport aux paramètres, en utilisant l'approche naïve et l'algorithme de *backprop*.

→ **L'approche naïve consiste en une série de calculs indépendants :** chaque couche doit dériver la loss finale, exprimée avec tous les paramètres utilisés dans le réseau, par rapport aux siens. Une partie de ce qu'elle calcule l'est donc aussi par les couches qui la suivent, **avec finalement une complexité en $O(n_c^2)$ où n_c est le nombre de couches du réseau** : on répète n_c fois un nombre d'opérations borné par $p*n_c$, où p est le nombre de paramètres maximum par couche.

→ **En *backprop*, on prend le réseau à rebours**. D'abord, la couche de sortie calcule ses dérivées par rapport à la loss finale et transmet l'erreur à corriger en arrière. Le processus recommence couche par couche, jusqu'à la première qui reçoit l'erreur calculée par la deuxième. **Ce qui est déjà connu est transmis et n'a pas à être recalculé : on obtient une complexité en $O(n_c)$ en réalisant n_c fois un maximum de p dérivations.**

13 Quel critère doit respecter l'architecture du réseau pour permettre la backpropagation ?

L'algorithme de backpropagation nécessite que toutes les transformations du réseau soit composées de fonctions dérivables.

On termine par des considérations calculatoires, même si PyTorch s'en occupe pour nous.

14 La fonction SoftMax et la loss cross-entropy sont souvent utilisées ensemble et leur gradient est très simple. Montrez que la loss se simplifie en

$$l = -\sum_i y_i \tilde{y}_i + \log(\sum_i e^{\tilde{y}_i})$$

En combinant la loss cross-entropy $l(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$ et la fonction SoftMax $\hat{y}_i = \frac{\exp(\tilde{y}_i)}{\sum_{i=1}^{n_y} \exp(\tilde{y}_i)}$:

$$l = -\sum_i y_i \log\left(\frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}}\right)$$

Distribution du log

$$= -\sum_i y_i \left(\log(e^{\tilde{y}_i}) - \log(\sum_i e^{\tilde{y}_i})\right)$$

Simplification log(exp(x))

$$= -\sum_i y_i (\tilde{y}_i - \log(\sum_i e^{\tilde{y}_i}))$$

Distribution du facteur sur la parenthèse

$$= -\sum_i y_i \tilde{y}_i + \sum_i y_i \log(\sum_j e^{\tilde{y}_j})$$

$\sum_i y_i = 1$ à cause du one-hot encoding

$$= -\sum_i \boldsymbol{y_i \tilde{y}_i} + \mathbf{1} * \log(\sum_i \boldsymbol{e^{\tilde{y}_i}}) \quad \square$$

15 Ecrire le gradient $\nabla_{\tilde{y}} l$ de la loss (cross-entropy) par rapport à la sortie intermédiaire $\tilde{y}_i$.

Note : les gradients pour les questions 15, 16, 17 ont toujours la même taille que les vecteurs d'origine.

$$\frac{\partial l}{\partial \tilde{y}_i} = \frac{\partial(-\sum_i y_i \tilde{y}_i + log(\sum_i e^{\tilde{y}_i}))}{\partial \tilde{y}_i}$$

On applique juste les règles de dérivation.

$$= -\frac{\partial(\sum_i y_i \tilde{y}_i)}{\partial \tilde{y}_i} + \frac{(\sum_i e^{\tilde{y}_i})'}{\sum_j e^{\tilde{y}_j}}$$

$$= -y_i + \frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}} = SoftMax(\tilde{y}_i) - y_i = \hat{y}_i - y_i$$

donc

$$\nabla_{\tilde{y}} \boldsymbol{l} = \begin{bmatrix} \frac{\partial l}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial l}{\partial \tilde{y}_{n_y}} \end{bmatrix} = \begin{bmatrix} \hat{y}_1 - y_1 \\ \vdots \\ \hat{y}_{n_y} - y_{n_y} \end{bmatrix} = \hat{y} - y.$$

16 **En utilisant la backpropagation, écrire le gradient de la loss par rapport aux poids de la couche de sortie $\nabla_{W_y} l$. Notez que l'écriture de ce gradient utilise $\nabla_{\tilde{y}} l$. Faire de même pour $\nabla_{b_y} l$.**

$$\frac{\partial \tilde{y}_k}{\partial W_{ij}^y} = \frac{\partial(\sum_{l=1}^{n_h} W_{y,kl} \cdot h_l + b_{y,k})}{\partial W_{y,ij}} = \begin{cases} h_j \text{ si } k = i \\ 0 \text{ si } k \neq i \end{cases}$$

$$\frac{\partial l}{\partial W_{y,ij}} = \sum_k \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = \frac{\partial l}{\partial \tilde{y}_i} \cdot h_j$$

Chain rule puis simplification

$$\frac{\partial \tilde{y}_k}{\partial b_{y,i}} = \frac{(\sum_{l=1}^{n_h} W_{y,kl} \cdot h_l + b_{y,k})}{\partial b_{y,i}} = \begin{cases} 1 \text{ si } k = i \\ 0 \text{ si } k \neq i \end{cases}$$

$$\frac{\partial l}{\partial b_{y,i}} = \sum_k \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial b_{y,i}} = \frac{\partial l}{\partial \tilde{y}_i}$$

$$\nabla_{W_y} l = \begin{bmatrix} \frac{\partial l}{\partial W_{y,11}} & \cdots & \frac{\partial l}{\partial W_{y,1n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial W_{y,n_y 1}} & \cdots & \frac{\partial l}{\partial W_{y,n_y n_h}} \end{bmatrix} = \begin{bmatrix} \frac{\partial l}{\partial \tilde{y}_1} \cdot h_1 & \cdots & \frac{\partial l}{\partial \tilde{y}_1} \cdot h_{n_h} \\ \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial \tilde{y}_{n_y}} \cdot h_1 & \cdots & \frac{\partial l}{\partial \tilde{y}_{n_y}} \cdot h_{n_h} \end{bmatrix}$$

Donc $\nabla_{W_y} l = \nabla_{\tilde{y}} l \cdot h^T$.

On applique la même logique : $\nabla_{b_y} l = \nabla_{\tilde{y}} l$.

17 **Calculer les autres gradients : $\nabla_{\tilde{h}} l, \nabla_{W_h} l, \nabla_{b_h} l$.**

$\nabla_{\tilde{h}} l$: gradient de la sortie d'une couche avant activation

Gradient post-activation

$$\frac{\partial \tilde{y}_k}{\partial h_i} = \frac{\partial(\sum_{l=1}^{n_h} W_{y,kl} \cdot h_l + b_{y,k})}{\partial h_i} = W_{y,ki} \quad donc \quad \frac{\partial l}{\partial h_i} = \sum_k \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial h_i} = \sum_k \frac{\partial l}{\partial \tilde{y}_k} \cdot W_{y,ki}$$

Chain rule puis simplification

…et un pas en arrière pour la pré-activation

$$\frac{\partial h_k}{\partial \tilde{h}_i} = \frac{\partial(tanh(\tilde{h}_k))}{\partial \tilde{h}_i} = \begin{cases} 1 - tanh^2(\tilde{h}_k) = 1 - tanh^2(\text{arctanh}(h_k)) = 1 - {h_i}^2 & si\ k = i \\ 0 & si\ k \neq i \end{cases}$$

$$\frac{\partial l}{\partial \tilde{h}_i} = \sum_k \frac{\partial l}{\partial h_k} \frac{\partial h_k}{\partial \tilde{h}_i} = (1 - {h_i}^2) \sum_k \frac{\partial l}{\partial \tilde{y}_k} \cdot W_{y,ki}$$

Idem

Donc $$\nabla_{\tilde{h}} \boldsymbol{l} = \begin{bmatrix} \frac{\partial l}{\partial \tilde{h}_1} \\ \vdots \\ \frac{\partial l}{\partial \tilde{h}_{n_h}} \end{bmatrix} = \begin{bmatrix} (1 - \boldsymbol{h_1}^2) \sum_k \frac{\partial l}{\partial \tilde{y}_k} \cdot \boldsymbol{W_{y,k1}} \\ \vdots \\ (1 - \boldsymbol{h_{n_h}}^2) \sum_k \frac{\partial l}{\partial \tilde{y}_k} \cdot \boldsymbol{W_{y,kn_h}} \end{bmatrix} = \boldsymbol{W_y^T} \cdot \boldsymbol{\nabla_{\tilde{y}} l} \odot (\boldsymbol{1} - \boldsymbol{h_1}^2)$$

$\nabla_{W_h} l$: gradient des poids pour cette couche

$$\frac{\partial \tilde{h}_k}{\partial W_{h,ij}} = \frac{\partial(\sum_{l=1}^{n_x} W_{h,kl} \cdot x_l + b_{h,k})}{\partial W_{h,ij}} = \begin{cases} x_j \text{ si } k = i \\ 0 \text{ si } k \neq i \end{cases}$$

$$\frac{\partial l}{\partial W_{h,ij}} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W_{h,ij}} = \frac{\partial l}{\partial \tilde{h}_i} \cdot x_j$$

Chain rule puis simplification

$$\nabla_{W_h} l = \begin{bmatrix} \frac{\partial l}{\partial W_{h,11}} & \cdots & \frac{\partial l}{\partial W_{h,1n_x}} \\ \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial W_{h,n_h 1}} & \cdots & \frac{\partial l}{\partial W_{h,n_h n_x}} \end{bmatrix} = \begin{bmatrix} \frac{\partial l}{\partial \tilde{h}_1} \cdot x_1 & \cdots & \frac{\partial l}{\partial \tilde{h}_1} \cdot x_{n_x} \\ \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial \tilde{h}_{n_h}} \cdot x_1 & \cdots & \frac{\partial l}{\partial \tilde{h}_{n_h}} \cdot x_{n_x} \end{bmatrix}$$

Donc $\nabla_{W_h} l = \nabla_{\tilde{h}} l \cdot x^T$

$\nabla_{b_h} l$: gradient du biais pour cette couche

$$\frac{\partial \tilde{h}_k}{\partial b_{h,i}} = \frac{\partial(\sum_{l=1}^{n_x} W_{h,kl} \cdot x_l + b_{h,k})}{\partial b_{h,i}} = \begin{cases} 1 \text{ si } k = i \\ 0 \text{ si } k \neq i \end{cases}$$

$$\frac{\partial l}{\partial b_{h,i}} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial b_{h,i}} = \frac{\partial l}{\partial \tilde{h}_i}$$

Donc $\nabla_{b_h} l = \nabla_{\tilde{h}} l$

Interprétation des résultats

La théorie développée plus haut peut être facilement mise en œuvre sous Python. L'objectif de ce TME était d'apprendre à automatiser les calculs avec PyTorch ; on code donc d'abord une descente de gradient entièrement manuelle avant de remplacer ce qui peut l'être morceau par morceau.

Entraînement sur les données Circle

On essaie d'abord d'entraîner un modèle sur les données Circle, qui sont composées de deux classes concentriques. Cette disposition des données empêche toute discrimination des classes avec des frontières linéaires ; le réseau de neurones est donc tout indiqué. Seul le *kernel trick* d'un classifieur SVM (on y vient) a du potentiel pour le concurrencer.

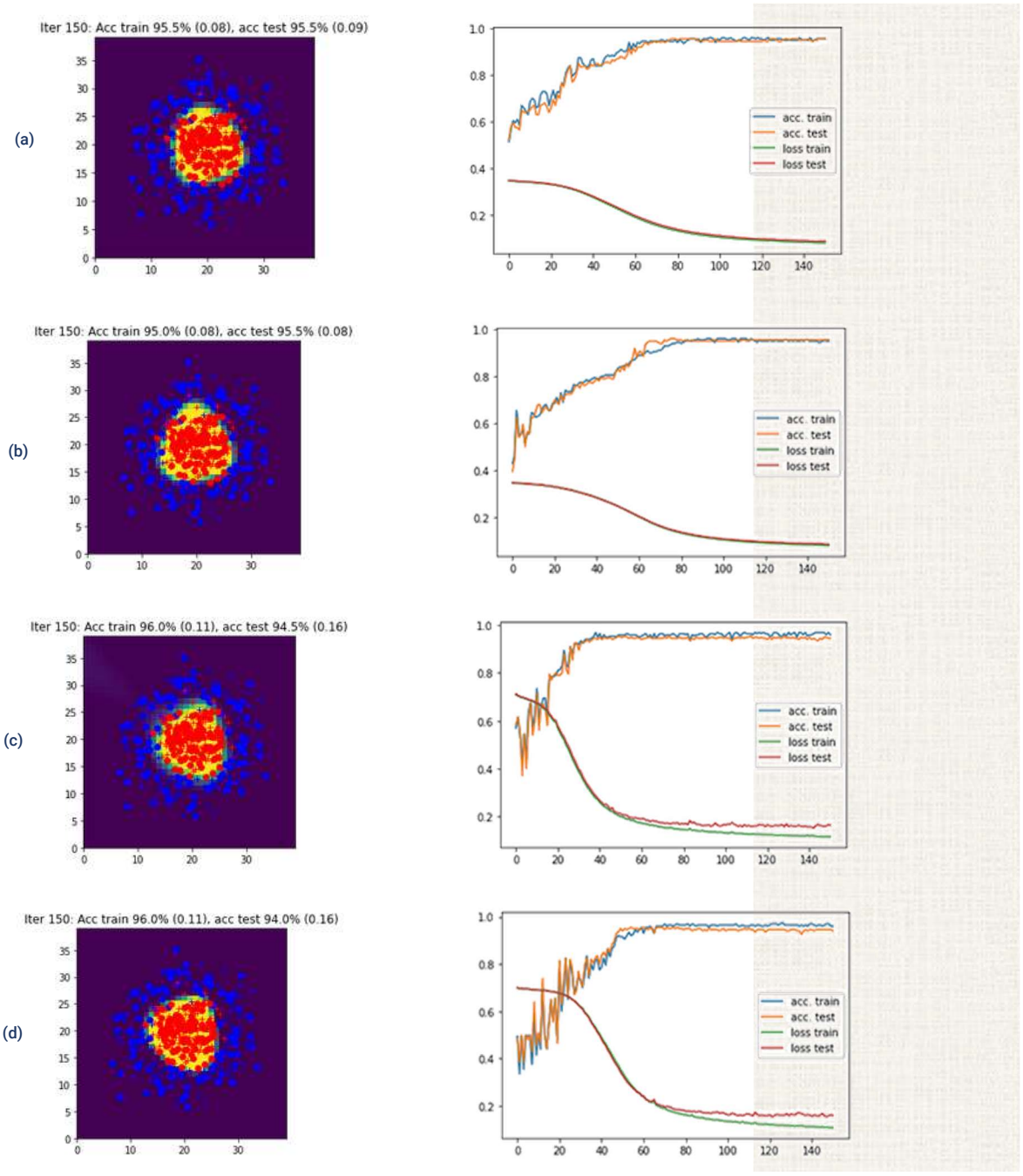


Figure 1 – Comparaison entre les résultats pour la version manuelle (a) et divers degrés d'automatisation de la descente de gradient : usage d'`Autograd` *(b), ajout de couches* `torch.nn` *(c), utilisation de l'optimiseur* `SGD` *(d). η = 0.1.*

On présente les résultats des exercices successifs avec 150 itérations et un pas d'apprentissage η toujours égal à 0.1. Nous n'avons pas eu de problème de divergence ou de minimum local (même si elle n'est pas linéaire, la frontière à repérer est simple). Les résultats sont toujours très bons avec toujours environ 95% d'accuracy en train comme en test, et l'automatisation se fait sans entrave.

Impact de l'automatisation

L'usage des couches `torch.nn` en (c) et (d) modifie un peu le comportement du modèle. En train comme en test, on constate plus d'instabilité qu'en (a) et (b) dans la loss des premières itérations, et une convergence plus rapide (on obtient un score quasi-parfait en moins de 40-50 itérations contre 60-70 avant). Les paramètres n'ont pourtant pas changé.
Plus important, on constate une réaugmentation de la loss en test sur les dernières itérations, qui indique un surapprentissage – car elle remonte alors que la loss en train diminue. On n'observait pas cela avant. Etant donné qu'on a convergé tôt, on peut d'ailleurs penser à arrêter l'apprentissage avant 100 itérations pour réduire ce phénomène (*early stopping*).

La raison de toutes ces observations est probablement l'initialisation des paramètres des couches `torch.nn` qui diffère de notre initialisation manuelle. Tout demeure semblable par ailleurs.

Justification des scores et impact de η

On ne décolle pas des 96% de bonne classification. Cela s'explique par le bruit autour de la frontière des classes. Quelques points sont dispersés de part et d'autre et ne peuvent pas être correctement classés si la frontière reste simple.
Pour tenter de faire mieux, on essaie l'algorithme correspondant à la Figure 1(d) (le plus haut degré d'automatisation) avec un η plus petit : peut-on améliorer la précision ? On veille aussi à étendre la durée de l'apprentissage pour que le modèle ait le temps de converger.

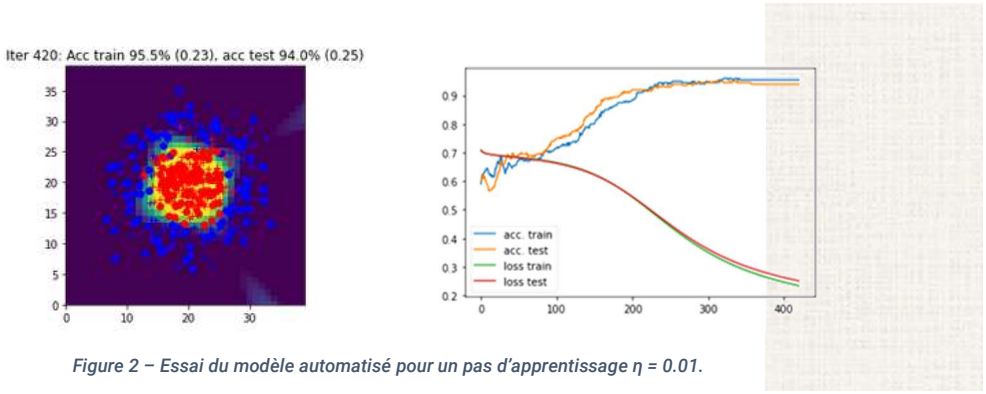


Figure 2 – Essai du modèle automatisé pour un pas d'apprentissage η = 0.01.

On n'a pas de différence majeure, et pour un η encore plus petit, l'apprentissage ne se fait pas bien. C'est donc bien le plus haut score possible avec cette architecture.

Bonus : SVM

On souhaite comparer les résultats des réseaux de neurones avec ceux d'une SVM.
→ En première approche, on essaie un classifieur `LinearSVC` de sklearn qui ne parvient pas à séparer les données, puisque la frontière n'est pas du tout linéaire.

→ On essaie alors des *kernel trick* sur des classifieurs `SVC` : on les paramètre avec `kernel='rbf'` (gaussien), `'sigmoid'`, `'poly'`. Le concept du *kernel trick* est de projeter les données dans un espace de plus haute dimension, et de les séparer linéairement dans cet espace.

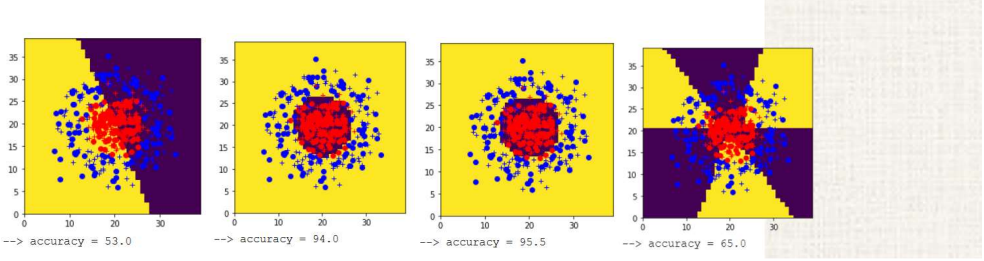


Figure 3 – Frontières de décision pour des SVM à noyau linéaire, gaussien, polynomial (degré 2) ou sigmoïde.

La meilleure performance est obtenue avec un noyau gaussien ou polynomial de degré 2.
De fait, le concept des distributions gaussiennes ou des polynômes de degré pair est bien adapté à la notion de centre et de périphérie. En 2D, on l'intuite facilement : on peut tracer une seule ligne horizontale pour isoler les images des points proches de 0 de tout le reste.

→ Dernière expérience : on fait varier le paramètre de régularisation C des SVM à noyau gaussien ou polynomial. Ce paramètre a de l'impact sur le score en test, car il régule la largeur de la marge pour la frontière de décision – ou en d'autres termes le degré de tolérance à la mauvaise classification. S'il est faible, la frontière sera lisse, peut-être trop ; s'il est élevé, elle aura plus de convolutions pour éviter un maximum d'erreurs, ce qui entraîne un surapprentissage.

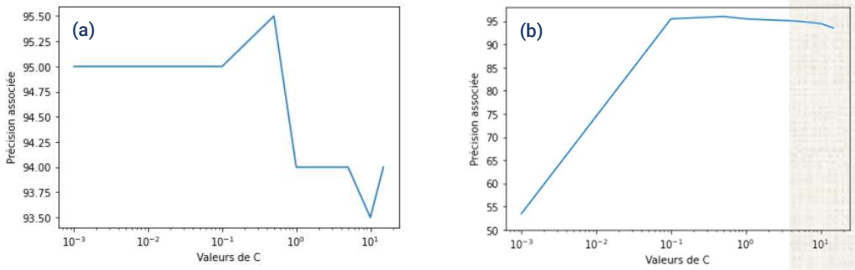


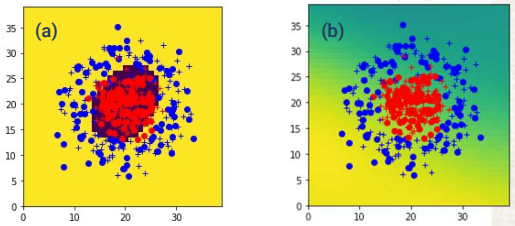
Figure 4 – Impact du paramètre de régularisation C sur une SVM à noyau gaussien (a) ou polynomial (b).

Il faut donc un juste milieu qu'on atteint pour C = 0.1 pour les deux.

→ Alors, aurait-on mieux fait d'utiliser des SVM sur les données Circle ? Peut-être que oui. Les scores sont similaires et il y a moins de paramètres (seulement des positions x et y à pondérer, contre un paramétrage de plusieurs neurones dans le réseau).

Dans le cas général, les réseaux de neurones ont la priorité. Ils peuvent de fait apprendre des frontières non-linéaires d'une variabilité infinie (là où celles des SVM ont une forme conditionnée par le type de kernel). Mais si le kernel est adapté, la SVM reste une alternative fiable et plus rapide que le Deep Learning du fait de la restriction des paramètres à optimiser. Pour preuve :

Figure 5 – Frontière de décision sur les données Circle, pour une SVM à noyau gaussien (a) vs. un réseau de neurones (b), après seulement dix itérations.



Note : la frontière de décision SVM est « dure » dès le départ. Juste ou pas, elle ne dépend que d'une seule transformation sur les coordonnées x et y des points. Celle choisie par le réseau de neurones est conditionnée par plus de paramètres ; et sur un plan, cela se traduit par une zone de transition.

Application aux données MNIST

Une fois qu'on en a fini avec les données Circle, on teste aussi les données MNIST (classification de chiffres manuscrits). Les scores obtenus sont convaincants avec 93% de précision en train comme en test.

```
Iter 0: Acc train 86.0% (0.50), acc test 86.6% (0.49)
Iter 10: Acc train 89.2% (0.36), acc test 89.1% (0.35)
Iter 20: Acc train 89.6% (0.34), acc test 89.7% (0.33)
Iter 30: Acc train 91.1% (0.29), acc test 91.0% (0.29)
Iter 40: Acc train 91.2% (0.28), acc test 91.7% (0.28)
Iter 50: Acc train 92.2% (0.26), acc test 92.2% (0.26)
Iter 60: Acc train 92.8% (0.24), acc test 92.7% (0.23)
Iter 70: Acc train 91.7% (0.27), acc test 91.8% (0.27)
Iter 80: Acc train 93.1% (0.23), acc test 93.2% (0.22)
Iter 90: Acc train 93.3% (0.22), acc test 93.4% (0.22)
```

Le réseau peut donc s'adapter à des données image. Mais remarquons qu'il les traite à plat, comme des vecteurs. Dans le TME suivant, nous apprendrons à prendre en compte la localité de l'information (et donc les patterns formés par des pixels voisins) avec les réseaux convolutionnels.

Conclusion et remise en contexte

On a appris à entraîner un réseau de neurones sur des données basiques et à automatiser cet entraînement. Les résultats étaient prometteurs sur deux petits datasets, Circle et MNIST, dont on a considéré les enregistrements sous la forme de *vecteurs de caractéristiques*.
Il n'y avait donc aucune adaptation spéciale au domaine de l'image.

La comparaison avec les SVM a permis de comprendre les avantages et inconvénients des réseaux de neurones : par rapport à une SVM, la présence des non-linéarités en Deep Learning permet de complexifier les frontières de décision à l'envi – au prix d'une explosion du nombre de paramètres. Cela permet de traiter des cas où la classification SVM n'est initialement pas applicable.

Le modèle appris cette semaine est très simple et n'est pas efficace pour toutes les applications, notamment le traitement de données multidimensionnelles ; c'est parce qu'il n'a qu'une couche, mais c'est aussi dû à son fonctionnement *fully-connected* (voir p. 2). La prochaine séance permettra donc d'introduire les réseaux convolutionnels qui y sont spécialement dédiés.