Introduction au Deep Learning (notes de cours)

Romain Tavenard

16 novembre 2023

TABLE DES MATIÈRES

par Romain Tavenard

Ce document sert de notes de cours pour un cours dispensé à l'Université de Rennes 2 (France) et à l'EDHEC Lille (France).

Le cours traite des bases des réseaux de neurones pour la classification et la régression sur des données tabulaires (y compris les algorithmes d'optimisation pour les perceptrons multicouches), les réseaux de neurones convolutifs pour la classification d'images (y compris les notions d'apprentissage par transfert) et la classification / prévision de séquences.

Les séances de travaux pratiques de ce cours utiliseront `keras`, tout comme ces notes de cours.

NB : ces notes ont été traduites vers le français de manière semi-automatique, n'hésitez pas à vous référer à la version anglaise en cas de doute.

CHAPITRE 1

INTRODUCTION

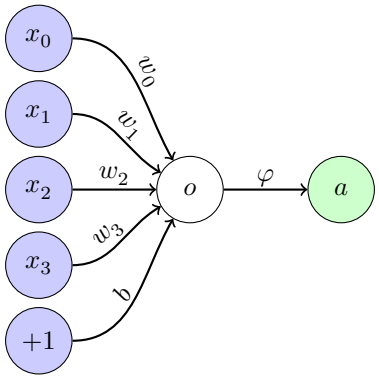
Dans ce chapitre d'introduction, nous allons présenter un premier réseau neuronal appelé le Perceptron. Ce modèle est un réseau neuronal constitué d'un seul neurone, et nous l'utiliserons ici pour introduire des concepts-clés que nous détaillerons plus tard dans le cours.

1.1 Un premier modèle : le perceptron

Dans la terminologie des réseaux de neurones, un neurone est une fonction paramétrée qui prend un vecteur **x** en entrée et sort une valeur unique a comme suit :

$$a = \varphi(\underbrace{\mathbf{wx} + b}_{o}),$$

où les paramètres du neurone sont ses poids stockés dans **w**. et un terme de biais b, et φ est une fonction d'activation qui est choisie a priori (nous y reviendrons plus en détail plus tard dans le cours) :



Un modèle constitué d'un seul neurone est appelé perceptron.

1.2 Optimisation

Les modèles présentés dans ce document ont pour but de résoudre des problèmes de prédiction dans lesquels l'objectif est de trouver des valeurs de paramètres « suffisamment bonnes » pour le modèle en jeu compte tenu de données observées.

Le problème de la recherche de telles valeurs de paramètres est appelé optimisation. L'apprentissage profond (ou *deep learning*) fait un usage intensif d'une famille spécifique de stratégies d'optimisation appelée **descente gradiente**.

1.2.1 Descente de gradient

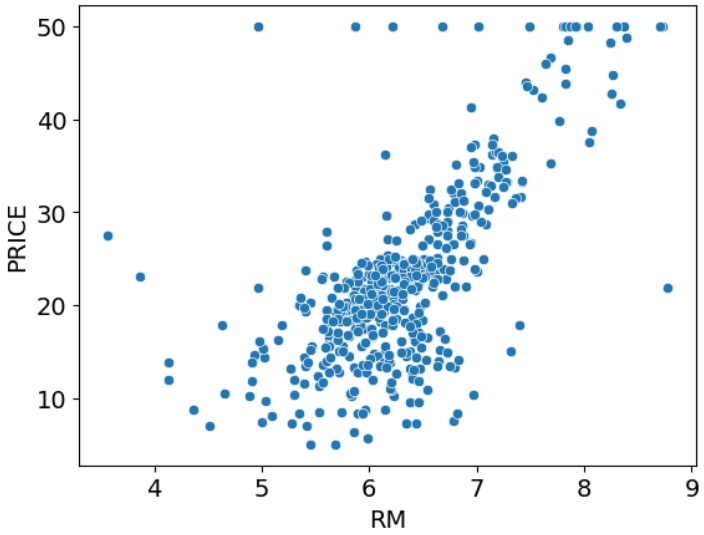
Pour se faire une idée de la descente de gradient, supposons que l'on nous donne le jeu de données suivant sur les prix de l'immobilier :

```
import pandas as pd

boston = pd.read_csv("../data/boston.csv")[["RM", "PRICE"]]
boston
```

```
        RM  PRICE
0    6.575   24.0
1    6.421   21.6
2    7.185   34.7
3    6.998   33.4
4    7.147   36.2
..     ...    ...
501  6.593   22.4
502  6.120   20.6
503  6.976   23.9
504  6.794   22.0
505  6.030   11.9

[506 rows x 2 columns]
```

Dans notre cas, nous essaierons (pour commencer) de prédire la valeur cible `"PRICE"` de ce jeu de données, qui est la valeur médiane des maisons occupées par leur propriétaire en milliers de dollars en fonction du nombre moyen de pièces par logement `"RM"` :

```
sns.scatterplot(data=boston, x="RM", y="PRICE");
```



Une courte note sur ce modèle

Dans la terminologie du Perceptron, ce modèle :
— n'a pas de fonction d'activation (*i.e.* φ est la fonction d'identité)
— n'a pas de biais (*i.e.* b est fixé à 0, il n'est pas appris)

Supposons que nous ayons une approche naïve dans laquelle notre modèle de prédiction est linéaire sans biais, c'est-à-dire que pour une entrée donnée x_i la sortie prédite est calculée comme suit :

$$\hat{y_i} = wx_i$$

où w est le seul paramètre de notre modèle.

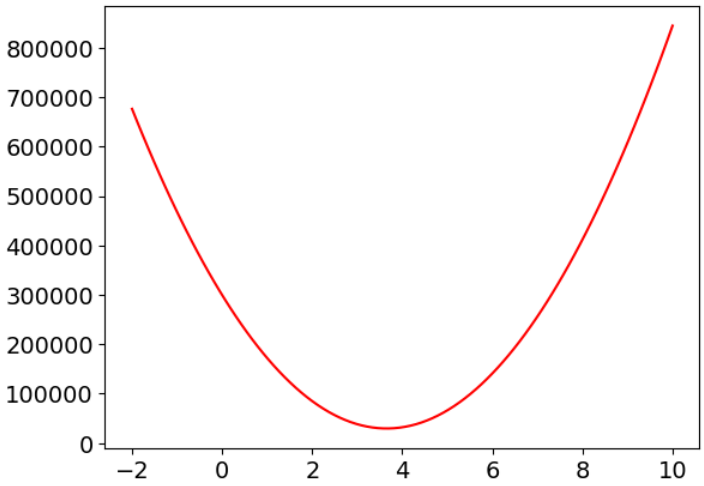
Supposons en outre que la quantité que nous cherchons à minimiser (notre objectif, également appelé fonction de perte) est :

$$\mathcal{L}(w) = \sum_i (\hat{y_i} - y_i)^2$$

où y_i est la valeur cible associée au i-ème échantillon de jeu de données.

Examinons cette quantité en fonction de w :

```
import numpy as np

def loss(w, x, y):
    w = np.array(w)
    return np.sum(
        (w[:, None] * x.to_numpy()[None, :] - y.to_numpy()[None, :]) ** 2,
        axis=1
    )

w = np.linspace(-2, 10, num=100)

x = boston["RM"]
y = boston["PRICE"]
plt.plot(w, loss(w, x, y), "r-");
```



Ici, il semble qu'une valeur de w autour de 4 devrait être un bon choix. Cette méthode (générer de nombreuses valeurs pour le paramètre et calculer la perte pour chaque valeur) ne peut pas s'adapter aux modèles qui ont beaucoup de paramètres, donc nous allons donc essayer autre chose.

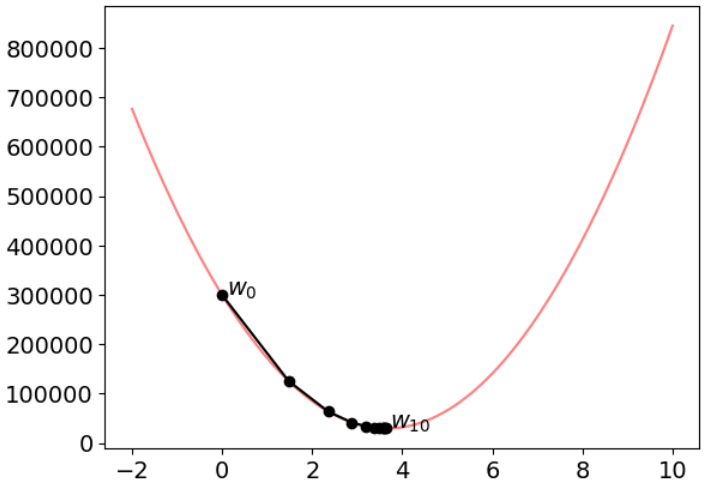
Supposons que nous ayons accès, à chaque fois que nous choisissons une valeur candidate pour w, à la fois à la perte $\mathcal{L}$ et aux informations sur la façon dont $\mathcal{L}$ varie, localement. Nous pourrions, dans ce cas, calculer une nouvelle valeur candidate pour w en nous déplaçant à partir de la valeur candidate précédente dans la direction de la descente la plus raide. C'est l'idée de base de l'algorithme de descente du gradient qui, à partir d'un candidat initial w_0, calcule itérativement de nouveaux candidats comme :

$$w_{t+1} = w_t - \rho \left. \frac{\partial \mathcal{L}}{\partial w} \right|_{w=w_t}$$

où ρ est un hyper-paramètre (appelé taux d'apprentissage) qui contrôle la taille des pas à effectuer, et $\left. \frac{\partial \mathcal{L}}{\partial w} \right|_{w=w_t}$ est le gradient de $\mathcal{L}$ par rapport à w, évalué en $w = w_t$. Comme vous pouvez le voir, la direction de la descente la plus raide est l'opposé de la direction indiquée par le gradient (et cela vaut aussi pour les paramètres vectoriels).

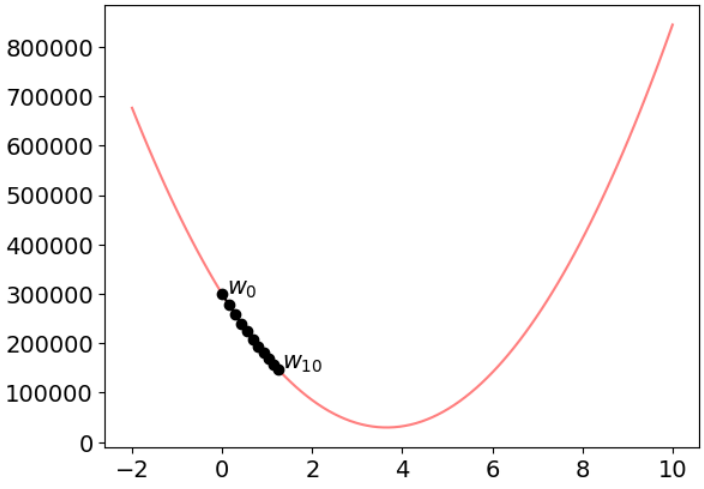
Ce processus est répété jusqu'à la convergence, comme l'illustre la figure suivante :

```
rho = 1e-5

def grad_loss(w_t, x, y):
    return np.sum(
        2 * (w_t * x - y) * x
    )


ww = np.linspace(-2, 10, num=100)
plt.plot(ww, loss(ww, x, y), "r-", alpha=.5);

w = [0.]
for t in range(10):
    w_update = w[t] - rho * grad_loss(w[t], x, y)
    w.append(w_update)

plt.plot(w, loss(w, x, y), "ko-")
plt.text(x=w[0]+.1, y=loss([w[0]], x, y), s="$w_{0}$")
plt.text(x=w[10]+.1, y=loss([w[10]], x, y), s="$w_{10}$");
```



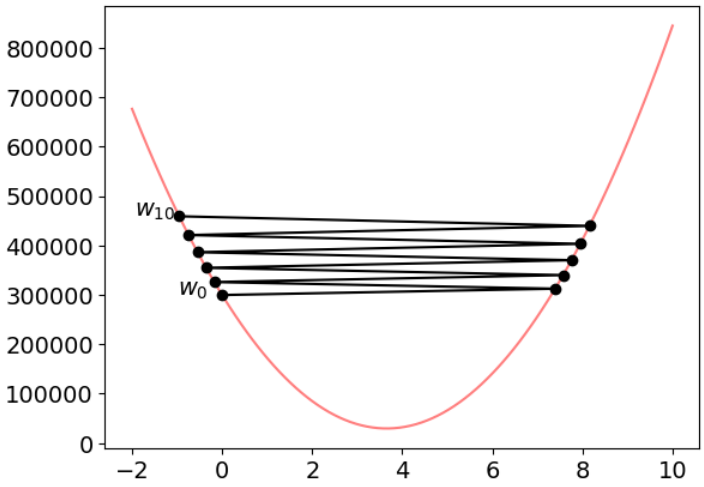
Qu'obtiendrions-nous si nous utilisions un taux d'apprentissage plus faible ?

```
rho = 1e-6

ww = np.linspace(-2, 10, num=100)
plt.plot(ww, loss(ww, x, y), "r-", alpha=.5);

w = [0.]
for t in range(10):
    w_update = w[t] - rho * grad_loss(w[t], x, y)
    w.append(w_update)

plt.plot(w, loss(w, x, y), "ko-")
plt.text(x=w[0]+.1, y=loss([w[0]], x, y), s="$w_{0}$")
plt.text(x=w[10]+.1, y=loss([w[10]], x, y), s="$w_{10}$");
```



Cela prendrait certainement plus de temps pour converger. Mais attention, un taux d'apprentissage plus élevé n'est pas toujours une bonne idée :

```
rho = 5e-5

ww = np.linspace(-2, 10, num=100)
plt.plot(ww, loss(ww, x, y), "r-", alpha=.5);

w = [0.]
for t in range(10):
    w_update = w[t] - rho * grad_loss(w[t], x, y)
    w.append(w_update)

plt.plot(w, loss(w, x, y), "ko-")
plt.text(x=w[0]-1., y=loss([w[0]], x, y), s="$w_{0}$")
plt.text(x=w[10]-1., y=loss([w[10]], x, y), s="$w_{10}$");
```



Vous voyez comment nous divergeons lentement parce que nos pas sont trop grands ?

1.3 Récapitulatif

Dans cette section, nous avons introduit :

- un modèle très simple, appelé le Perceptron : ce sera une brique de base pour les modèles plus avancés que nous détaillerons plus tard dans le cours, tels que :
  - le *Perceptron multi-couches*
  - les *architectures convolutionnelles*
  - les *architectures récurrentes*
- le fait qu'une tâche s'accompagne d'une fonction de perte à minimiser (ici, nous avons utilisé l'erreur quadratique moyenne pour notre tâche de régression), qui sera discutée dans *un chapitre dédié* ;
- le concept de descente de gradient pour optimiser la perte choisie sur le paramètre unique d'un modèle, et ceci sera étendu dans *notre chapitre sur l'optimisation*.

CHAPITRE 2

PERCEPTRONS MULTICOUCHES

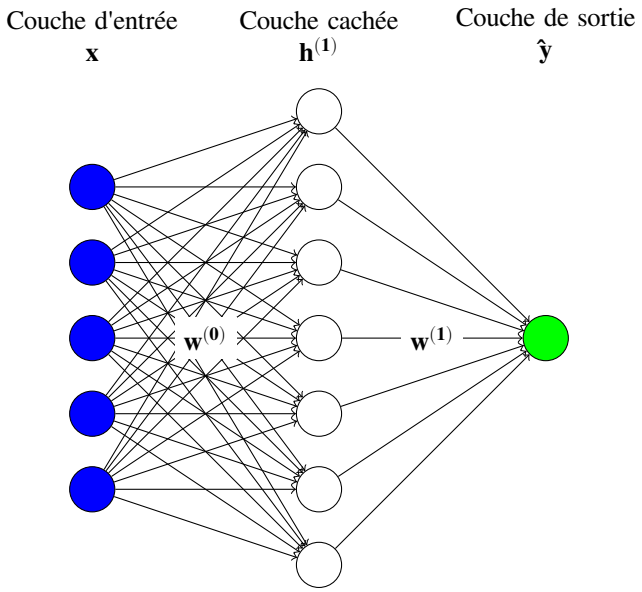
Dans le chapitre précédent, nous avons vu un modèle très simple appelé le perceptron. Dans ce modèle, la sortie prédite $\hat{y}$ est calculée comme une combinaison linéaire des caractéristiques d'entrée plus un biais :

$$\hat{y} = \sum_{j=1}^{d} x_j w_j + b$$

En d'autres termes, nous optimisions parmi la famille des modèles linéaires, qui est une famille assez restreinte.

2.1 Empiler des couches pour une meilleure expressivité

Afin de couvrir un plus large éventail de modèles, on peut empiler des neurones organisés en couches pour former un modèle plus complexe, comme le modèle ci-dessous, qui est appelé modèle à une couche cachée, car une couche supplémentaire de neurones est introduite entre les entrées et la sortie :



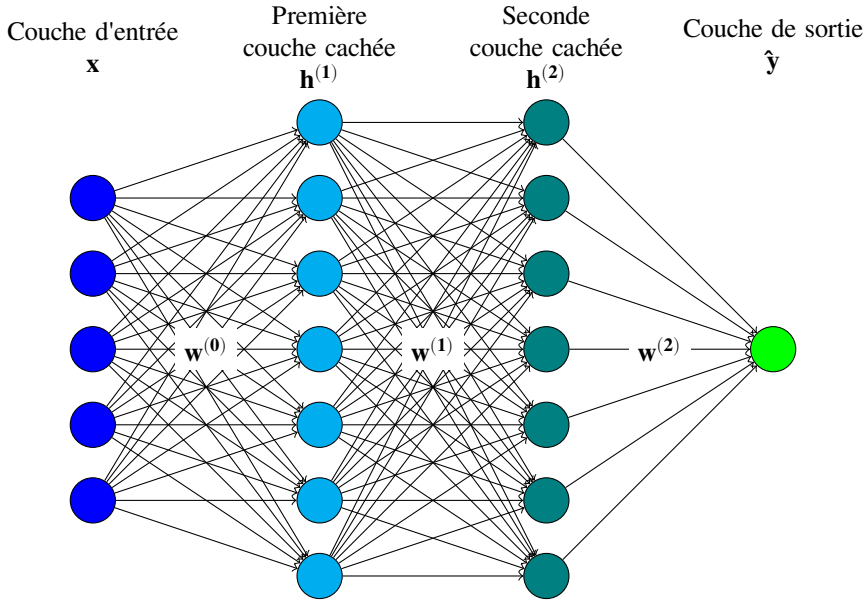
La question que l'on peut se poser maintenant est de savoir si cette couche cachée supplémentaire permet effectivement de couvrir une plus grande famille de modèles. C'est à cela que sert le théorème d'approximation universelle ci-dessous.

Théorème d'approximation universelle

Le théorème d'approximation universelle stipule que toute fonction continue définie sur un ensemble compact peut être approchée d'aussi près que l'on veut par un réseau neuronal à une couche cachée avec activation sigmoïde.

En d'autres termes, en utilisant une couche cachée pour mettre en correspondance les entrées et les sorties, on peut maintenant approximer n'importe quelle fonction continue, ce qui est une propriété très intéressante. Notez cependant que le nombre de neurones cachés nécessaire pour obtenir une qualité d'approximation donnée n'est pas discuté ici. De plus, il n'est pas suffisant qu'une telle bonne approximation existe, une autre question importante est de savoir si les algorithmes d'optimisation que nous utiliserons convergeront *in fine* vers cette solution ou non, ce qui n'est pas garanti, comme discuté plus en détail dans *le chapitre dédié*.

En pratique, nous observons empiriquement que pour atteindre une qualité d'approximation donnée, il est plus efficace (en termes de nombre de paramètres requis) d'empiler plusieurs couches cachées plutôt que de s'appuyer sur une seule :



La représentation graphique ci-dessus correspond au modèle suivant :

$$\hat{y} = \varphi_{\text{out}}\left(\sum_i w_i^{(2)} h_i^{(2)} + b^{(2)}\right) \tag{2.1}$$

$$\forall i, h_i^{(2)} = \varphi\left(\sum_j w_{ij}^{(1)} h_j^{(1)} + b_i^{(1)}\right) \tag{2.2}$$

$$\forall i, h_i^{(1)} = \varphi\left(\sum_j w_{ij}^{(0)} x_j + b_i^{(0)}\right) \tag{2.3}$$

Pour être précis, les termes de biais $b_i^{(l)}$ ne sont pas représentés dans la représentation graphique ci-dessus.

De tels modèles avec une ou plusieurs couches cachées sont appelés **Perceptrons multicouches** (ou *Multi-Layer Perceptrons*, MLP).

2.2 Décider de l'architecture d'un MLP

Lors de la conception d'un modèle de perceptron multicouche destiné à être utilisé pour un problème spécifique, certaines quantités sont fixées par le problème en question et d'autres sont des hyper-paramètres du modèle.

Prenons l'exemple du célèbre jeu de données de classification d'iris :

```
import pandas as pd

iris = pd.read_csv("../data/iris.csv", index_col=0)
iris
```

```
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                  5.1               3.5                1.4               0.2
1                  4.9               3.0                1.4               0.2
2                  4.7               3.2                1.3               0.2
3                  4.6               3.1                1.5               0.2
4                  5.0               3.6                1.4               0.2
..                 ...               ...                ...               ...
145                6.7               3.0                5.2               2.3
146                6.3               2.5                5.0               1.9
147                6.5               3.0                5.2               2.0
148                6.2               3.4                5.4               2.3
149                5.9               3.0                5.1               1.8

     target
0         0
1         0
2         0
3         0
4         0
..      ...
145       2
146       2
147       2
148       2
149       2

[150 rows x 5 columns]
```

L'objectif ici est d'apprendre à déduire l'attribut « cible » (3 classes différentes possibles) à partir des informations contenues dans les 4 autres attributs.

La structure de ce jeu de données dicte :

- le nombre de neurones dans la couche d'entrée, qui est égal au nombre d'attributs descriptifs dans notre jeu de données (ici, 4), et
- le nombre de neurones dans la couche de sortie, qui est ici égal à 3, puisque le modèle est censé produire une probabilité par classe cible.

De manière plus générale, pour la couche de sortie, on peut être confronté à plusieurs situations :

- lorsqu'il s'agit de régression, le nombre de neurones de la couche de sortie est égal au nombre de caractéristiques à prédire par le modèle,
- quand il s'agit de classification
  - Dans le cas d'une classification binaire, le modèle aura un seul neurone de sortie qui indiquera la probabilité de la classe positive,

— dans le cas d'une classification multi-classes, le modèle aura autant de neurones de sortie que le nombre de classes du problème.

Une fois que ces nombres de neurones d'entrée / sortie sont fixés, le nombre de neurones cachés ainsi que le nombre de neurones par couche cachée restent des hyper-paramètres du modèle.

2.3 Fonctions d'activation

Un autre hyper-paramètre important des réseaux neuronaux est le choix de la fonction d'activation φ.

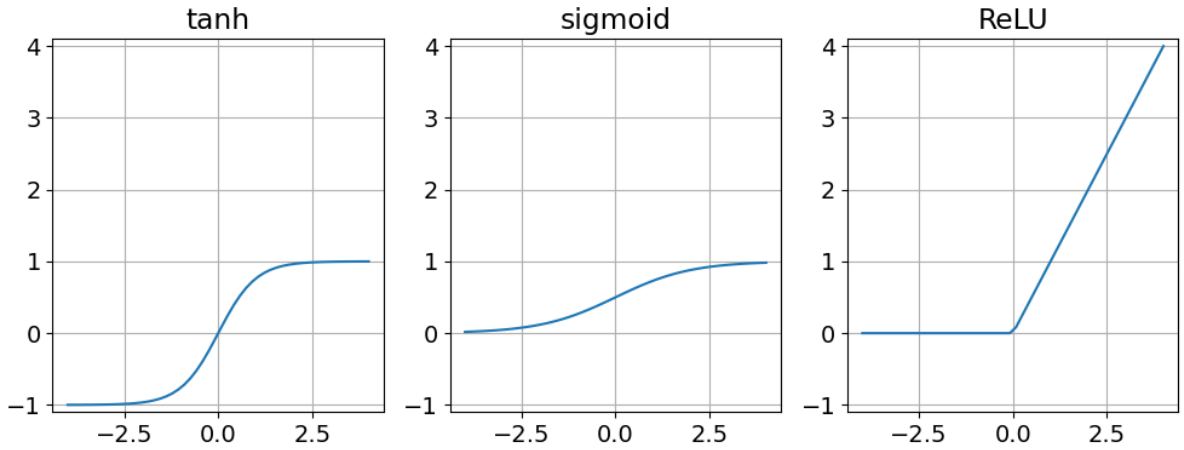
Il est important de noter que si nous utilisons la fonction identité comme fonction d'activation, quelle que soit la profondeur de notre MLP, nous ne couvrirons plus que la famille des modèles linéaires. En pratique, nous utiliserons donc des fonctions d'activation qui ont un certain régime linéaire mais qui ne se comportent pas comme une fonction linéaire sur toute la gamme des valeurs d'entrée.

Historiquement, les fonctions d'activation suivantes ont été proposées :

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

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

$$\text{ReLU}(x) = \begin{cases} x \text{ si } x > 0 \\ 0 \text{ sinon} \end{cases}$$



En pratique, la fonction ReLU (et certaines de ses variantes) est la plus utilisée de nos jours, pour des raisons qui seront discutées plus en détail dans *notre chapitre consacré à l'optimisation*.

2.3.1 Le cas particulier de la couche de sortie

Vous avez peut-être remarqué que dans la formulation du MLP fournie par l'équation (1), la couche de sortie possède sa propre fonction d'activation, notée φ_{out}. Cela s'explique par le fait que le choix de la fonction d'activation pour la couche de sortie d'un réseau neuronal est spécifique au problème à résoudre.

En effet, vous avez pu constater que les fonctions d'activation abordées dans la section précédente ne partagent pas la même plage de valeurs de sortie. Il est donc primordial de choisir une fonction d'activation adéquate pour la couche de sortie, de sorte que notre modèle produise des valeurs cohérentes avec les quantités qu'il est censé prédire.

Si, par exemple, notre modèle est censé être utilisé dans l'ensemble de données sur les logements de Boston dont nous avons parlé *dans le chapitre précédent*, l'objectif est de prédire les prix des logements, qui sont censés être des quantités non

négatives. Il serait donc judicieux d'utiliser ReLU (qui peut produire toute valeur positive) comme fonction d'activation pour la couche de sortie dans ce cas.

Comme indiqué précédemment, dans le cas de la classification binaire, le modèle aura un seul neurone de sortie et ce neurone produira la probabilité associée à la classe positive. Cette quantité devra se situer dans l'intervalle $[0, 1]$, et la fonction d'activation sigmoïde est alors le choix par défaut dans ce cas.

Enfin, lorsque la classification multi-classes est en jeu, nous avons un neurone par classe de sortie et chaque neurone est censé fournir la probabilité pour une classe donnée. Dans ce contexte, les valeurs de sortie doivent être comprises entre 0 et 1, et leur somme doit être égale à 1. À cette fin, nous utilisons la fonction d'activation softmax définie comme suit :

$$\forall i, \text{softmax}(o_i) = \frac{e^{o_i}}{\sum_j e^{o_j}}$$

où, pour tous les i, les o_i sont les valeurs des neurones de sortie avant application de la fonction d'activation.

2.4 Déclarer un MLP en `keras`

Pour définir un modèle MLP dans `keras`, il suffit d'empiler des couches. A titre d'exemple, si l'on veut coder un modèle composé de :

— une couche d'entrée avec 10 neurones,
— d'une couche cachée de 20 neurones avec activation ReLU,
— une couche de sortie composée de 3 neurones avec activation softmax,

le code sera le suivant :

```
import keras_core as keras
from keras.layers import Dense, InputLayer
from keras.models import Sequential

model = Sequential([
    InputLayer(input_shape=(10, )),
    Dense(units=20, activation="relu"),
    Dense(units=3, activation="softmax")
])

model.summary()
```

```
Using TensorFlow backend
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 20)                220

 dense_1 (Dense)             (None, 3)                 63

=================================================================
Total params: 283 (1.11 KB)
Trainable params: 283 (1.11 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Notez que `model.summary()` fournit un aperçu intéressant d'un modèle défini et de ses paramètres.

Exercice #1

En vous basant sur ce que nous avons vu dans ce chapitre, pouvez-vous expliquer le nombre de paramètres retournés par `model.summary()` ci-dessus ?

Solution

Notre couche d'entrée est composée de 10 neurones, et notre première couche est entièrement connectée, donc chacun de ces neurones est connecté à un neurone de la couche cachée par un paramètre, ce qui fait déjà $10 \times 20 = 200$ paramètres. De plus, chacun des neurones de la couche cachée possède son propre paramètre de biais, ce qui fait 20 paramètres supplémentaires. Nous avons donc 220 paramètres, tels que sortis par `model.summary()` pour la couche `"dense (Dense)"`.

De la même manière, pour la connexion des neurones de la couche cachée à ceux de la couche de sortie, le nombre total de paramètres est de $20 \times 3 = 60$ pour les poids plus 3 paramètres supplémentaires pour les biais.

Au total, nous avons $220 + 63 = 283$ paramètres dans ce modèle.

Exercice #2

Déclarez, en `keras`, un MLP avec une couche cachée composée de 100 neurones et une activation ReLU pour le jeu de données Iris présenté ci-dessus.

Solution

```
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=100, activation="relu"),
    Dense(units=3, activation="softmax")
])
```

Exercice #3

Même question pour le jeu de données sur le logement à Boston présenté ci-dessous (le but ici est de prédire l'attribut `PRICE` en fonction des autres).

Solution

```
model = Sequential([
    InputLayer(input_shape=(6, )),
    Dense(units=100, activation="relu"),
    Dense(units=1, activation="relu")
])
```

```
      RM     CRIM  INDUS    NOX   AGE    TAX  PRICE
0  6.575  0.00632   2.31  0.538  65.2  296.0   24.0
```

(suite sur la page suivante)

(suite de la page précédente)

```
1    6.421  0.02731   7.07  0.469  78.9  242.0   21.6
2    7.185  0.02729   7.07  0.469  61.1  242.0   34.7
3    6.998  0.03237   2.18  0.458  45.8  222.0   33.4
4    7.147  0.06905   2.18  0.458  54.2  222.0   36.2
..     ...      ...    ...    ...   ...    ...    ...
501  6.593  0.06263  11.93  0.573  69.1  273.0   22.4
502  6.120  0.04527  11.93  0.573  76.7  273.0   20.6
503  6.976  0.06076  11.93  0.573  91.0  273.0   23.9
504  6.794  0.10959  11.93  0.573  89.3  273.0   22.0
505  6.030  0.04741  11.93  0.573  80.8  273.0   11.9

[506 rows x 7 columns]
```

CHAPITRE 3

FONCTIONS DE COÛT

Nous avons maintenant présenté une première famille de modèles, qui est la famille MLP. Afin d'entraîner ces modèles (*i.e.* d'ajuster leurs paramètres pour qu'ils s'adaptent aux données), nous devons définir une fonction de coût (aussi appelée fonction de perte, ou *loss function*) à optimiser. Une fois cette fonction choisie, l'optimisation consistera à régler les paramètres du modèle de manière à la minimiser.

Dans cette section, nous présenterons deux fonctions de pertes standard, à savoir l'erreur quadratique moyenne (principalement utilisée pour la régression) et la fonction de perte logistique (utilisée en classification).

Dans ce qui suit, nous supposons connu un ensemble de données $\mathcal{D}$ composé de n échantillons annotés (x_i, y_i), et nous désignons la sortie du modèle :

$$\forall i, \hat{y}_i = m_\theta(x_i)$$

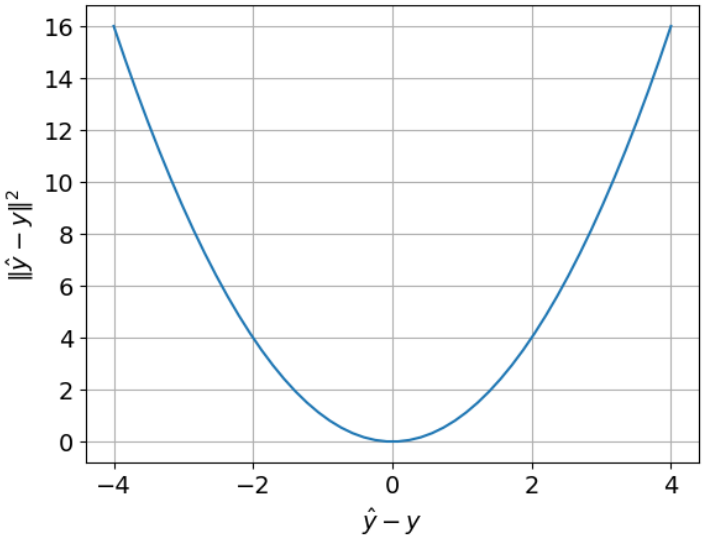
où m_θ est notre modèle et θ est l'ensemble de tous ses paramètres (poids et biais).

3.1 Erreur quadratique moyenne

L'erreur quadratique moyenne (ou *Mean Squared Error*, MSE) est la fonction de perte la plus couramment utilisée dans les contextes de régression. Elle est définie comme suit

$$\begin{aligned}\mathcal{L}(\mathcal{D}; m_\theta) &= \frac{1}{n}\sum_i \|\hat{y}_i - y_i\|^2 \\ &= \frac{1}{n}\sum_i \|m_\theta(x_i) - y_i\|^2\end{aligned}$$

Sa forme quadratique tend à pénaliser fortement les erreurs importantes :



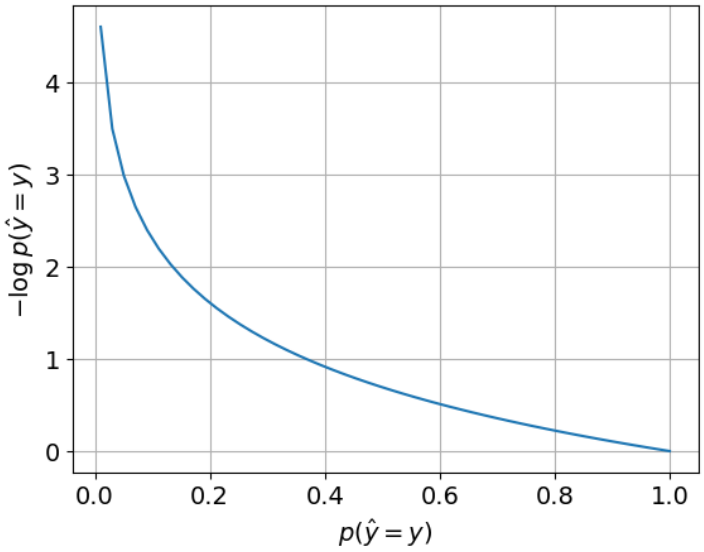
3.2 Perte logistique

La perte logistique est la fonction de perte la plus largement utilisée pour entraîner des réseaux neuronaux dans des contextes de classification. Elle est définie comme suit

$$\mathcal{L}(\mathcal{D}; m_\theta) = \frac{1}{n} \sum_i -\log p(\hat{y}_i = y_i; m_\theta)$$

où $p(\hat{y}_i = y_i; m_\theta)$ est la probabilité prédite par le modèle m_θ pour la classe correcte y_i.

Sa formulation tend à favoriser les cas où le modèle prédit la classe correcte avec une probabilité proche de 1, comme on peut s'y attendre :



CHAPITRE 4

OPTIMISATION

Dans ce chapitre, nous présenterons des variantes de la stratégie d'optimisation **de descente de gradient** et montrerons comment elles peuvent être utilisées pour optimiser les paramètres des réseaux de neurones.

Commençons par l'algorithme de base de la descente de gradient et ses limites.

Algorithm 1 (Descente de Gradient)

Entrée: Un jeu de données $\mathcal{D} = (X, y)$

1. Initialiser les paramètres θ du modèle
2. for $e = 1..E$
    1. for $(x_i, y_i) \in \mathcal{D}$
        1. Calculer la prédiction $\widehat{y}_i = m_\theta(x_i)$
        2. Calculer le gradient individuel $\nabla_\theta \mathcal{L}_i$
    2. Calculer le gradient total $\nabla_\theta \mathcal{L} = \frac{1}{n} \sum_i \nabla_\theta \mathcal{L}_i$
    3. Mettre à jour les paramètres θ à partir de $\nabla_\theta \mathcal{L}$

La règle de mise à jour typique pour les paramètres θ à l'itération t est

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \rho \nabla_\theta \mathcal{L}$$

où ρ est un hyper-paramètre important de la méthode, appelé le taux d'apprentissage (ou *learning rate*). La descente de gradient consiste à jour itérativement θ dans la direction de la plus forte diminution de la perte $\mathcal{L}$.

Comme on peut le voir dans l'algorithme précédent, lors d'un descente de gradient, les paramètres du modèle sont mis à jour une fois par *epoch*, ce qui signifie qu'un passage complet sur l'ensemble des données est nécessaire avant la mise à jour. Lorsque l'on traite de grands jeux de données, cela constitue une forte limitation, ce qui motive l'utilisation de variantes stochastiques.

4.1 Descente de gradient stochastique

L'idée derrière l'algorithme de descente de gradient stochastique (ou *Stochastic Gradient Descent*, SGD) est d'obtenir des estimations bon marché (au sens de la quantité de calculs nécessaires) pour la quantité

$$\nabla_\theta \mathcal{L}(\mathcal{D}; m_\theta) = \frac{1}{n} \sum_{(x_i, y_i) \in \mathcal{D}} \nabla_\theta \mathcal{L}(x_i, y_i; m_\theta)$$

où $\mathcal{D}$ est l'ensemble d'apprentissage. Pour ce faire, on tire des sous-ensembles de données, appelés *minibatchs*, et

$$\nabla_\theta \mathcal{L}(\mathcal{B}; m_\theta) = \frac{1}{b} \sum_{(x_i, y_i) \in \mathcal{B}} \nabla_\theta \mathcal{L}(x_i, y_i; m_\theta)$$

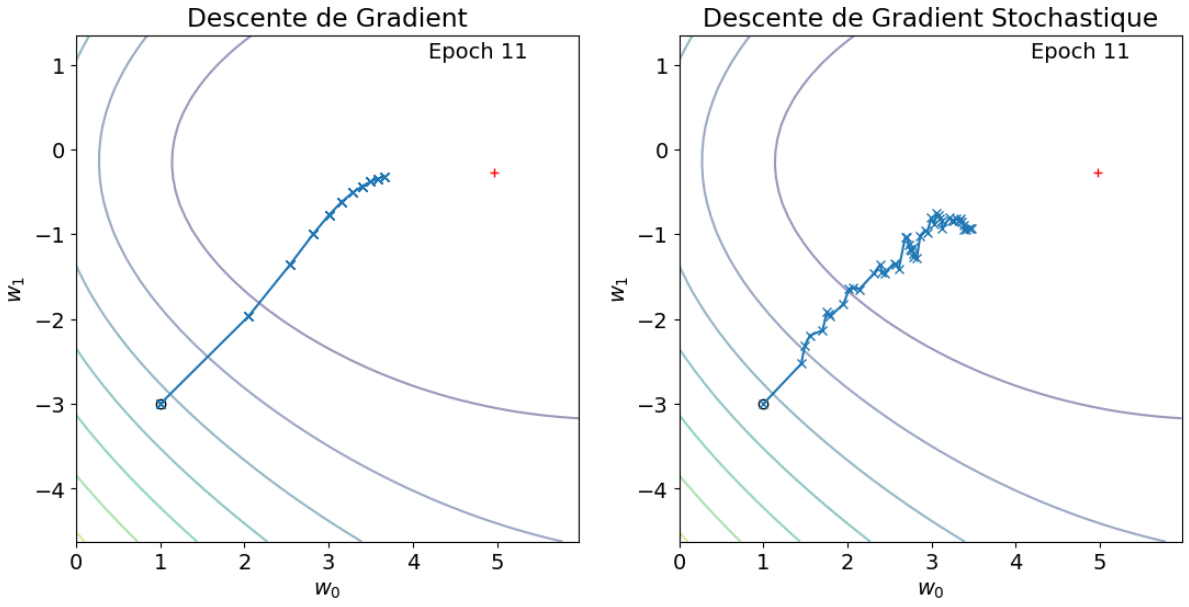
est utilisé comme estimateur de $\nabla_\theta \mathcal{L}(\mathcal{D}; m_\theta)$. Il en résulte l'algorithme suivant dans lequel les mises à jour des paramètres se produisent après chaque *minibatch*, c'est-à-dire plusieurs fois par *epoch*.

Algorithm 2 (Descente de gradient stochastique)

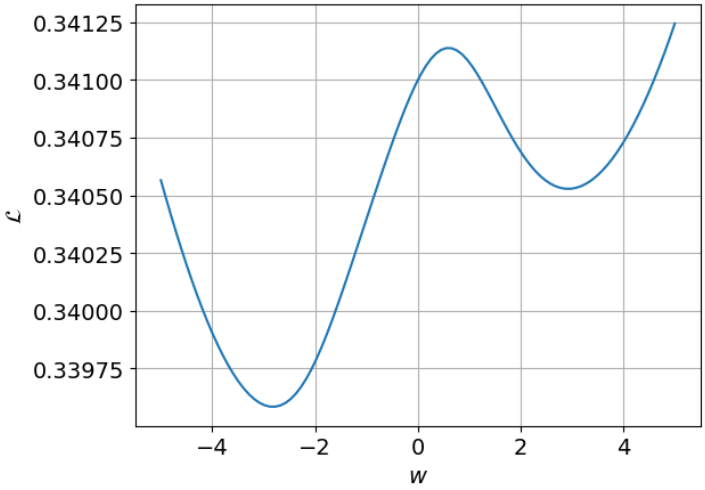
Input: A dataset $\mathcal{D} = (X, y)$

1. Initialiser les paramètres θ du modèle
2. for $e = 1..E$
    1. for $t = 1..n_{\text{minibatches}}$
        1. Tirer un échantillon aléatoire de taillle b dans $\mathcal{D}$ que l'on appelle *minibatch*
        2. for $(x_i, y_i) \in \mathcal{B}$
            1. Calculer la prédiction $\hat{y}_i = m_\theta(x_i)$
            2. Calculer le gradient individuel $\nabla_\theta \mathcal{L}_i$
        3. Calculer le gradient sommé sur le *minibatch* $\nabla_\theta \mathcal{L}_\mathcal{B} = \frac{1}{b} \sum_i \nabla_\theta \mathcal{L}_i$
        4. Mettre à jour les paramètres θ à partir de $\nabla_\theta \mathcal{L}_\mathcal{B}$

Par conséquent, lors de l'utilisation de SGD, les mises à jour des paramètres sont plus fréquentes, mais elles sont « bruitées » puisqu'elles sont basées sur une estimation du gradient par *minibatch* au lieu de s'appuyer sur le vrai gradient, comme illustré ci-dessous :



Outre le fait qu'elle implique des mises à jour plus fréquentes des paramètres, la SGD présente un avantage supplémentaire en termes d'optimisation, qui est essentiel pour les réseaux de neurones. En effet, comme on peut le voir ci-dessous, contrairement à ce que nous avions dans le cas du Perceptron, la perte MSE (et il en va de même pour la perte logistique) n'est plus convexe en les paramètres du modèle dès que celui-ci possède au moins une couche cachée :



La descente de gradient est connue pour souffrir d'optima locaux, et de tels fonctions de pertes constituent un problème sérieux pour la descente de gradient. D'un autre côté, la descente de gradient stochastique est susceptible de bénéficier d'estimations de gradient bruitées pour s'échapper des minima locaux.

4.2 Une note sur Adam

Adam [Kingma and Ba, 2015] est une variante de la méthode de descente de gradient stochastique. Elle diffère dans la règle de mise à jour des paramètres.

Tout d'abord, elle utilise ce qu'on appelle le momentum, qui consiste essentiellement à s'appuyer sur les mises à jour antérieures du gradient pour lisser la trajectoire dans l'espace des paramètres pendant l'optimisation. Une illustration interactive du momentum peut être trouvée dans [Goh, 2017].

L'estimation du gradient est remplacée par la quantité :

$$\mathbf{m}^{(t+1)} \leftarrow \frac{1}{1-\beta_1^t}\left[\beta_1 \mathbf{m}^{(t)} + (1-\beta_1)\nabla_\theta \mathcal{L}\right]$$

Lorsque β_1 est égal à zéro, nous avons $\mathbf{m}^{(t+1)} = \nabla_\theta \mathcal{L}$ et pour $\beta_1 \in]0, 1[$, $\mathbf{m}^{(t+1)}$ l'estimation courante du gradient utilise l'information sur les estimations passées, stockée dans $\mathbf{m}^{(t)}$.

Une autre différence importante entre SGD et la Adam consiste à utiliser un taux d'apprentissage adaptatif. En d'autres termes, au lieu d'utiliser le même taux d'apprentissage ρ pour tous les paramètres du modèle, le taux d'apprentissage pour un paramètre donné θ_i est défini comme :

$$\hat{\rho}^{(t+1)}(\theta_i) = \frac{\rho}{\sqrt{s^{(t+1)}(\theta_i) + \epsilon}}$$

où ϵ est une constante petite devant 1 et

$$s^{(t+1)}(\theta_i) = \frac{1}{1-\beta_2^t}\left[\beta_2 s^{(t)}(\theta_i) + (1-\beta_2)\left(\nabla_{\theta_i}\mathcal{L}\right)^2\right]$$

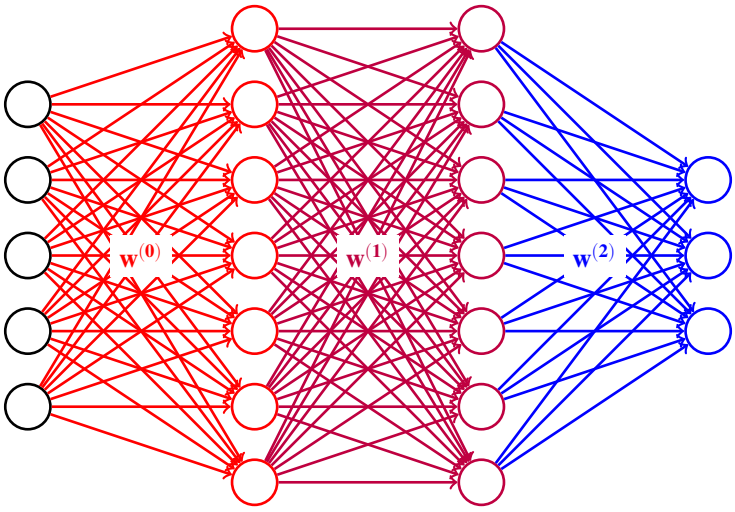
Ici aussi, le terme s utilise le momentum. Par conséquent, le taux d'apprentissage sera réduit pour les paramètres qui ont subi de grandes mises à jour dans les itérations précédentes.

Globalement, la règle de mise à jour d'Adam est la suivante :

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \hat{\rho}^{(t+1)}(\theta)\mathbf{m}^{(t+1)}$$

4.3 La malédiction de la profondeur

Considérons le réseau neuronal suivant :



et rappelons que, pour une couche donnée (ℓ), la sortie de la couche est calculée comme suit

$$a^{(\ell)} = \varphi(o^{(\ell)}) = \varphi(w^{(\ell-1)}a^{(\ell-1)})$$

où φ est la fonction d'activation pour la couche donnée (nous ignorons les termes de biais dans cet exemple simplifié).

Afin d'effectuer une descente de gradient (stochastique), les gradients de la perte par rapport aux paramètres du modèle doivent être calculés.

En utilisant la règle de la dérivation en chaîne, ces gradients peuvent être exprimés comme suit :

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w^{(2)}} &= \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial w^{(2)}} \\
\frac{\partial \mathcal{L}}{\partial w^{(1)}} &= \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial o^{(2)}} \frac{\partial o^{(2)}}{\partial w^{(1)}} \\
\frac{\partial \mathcal{L}}{\partial w^{(0)}} &= \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial o^{(2)}} \frac{\partial a^{(1)}}{\partial o^{(1)}} \frac{\partial o^{(1)}}{\partial w^{(0)}}
\end{aligned}$$

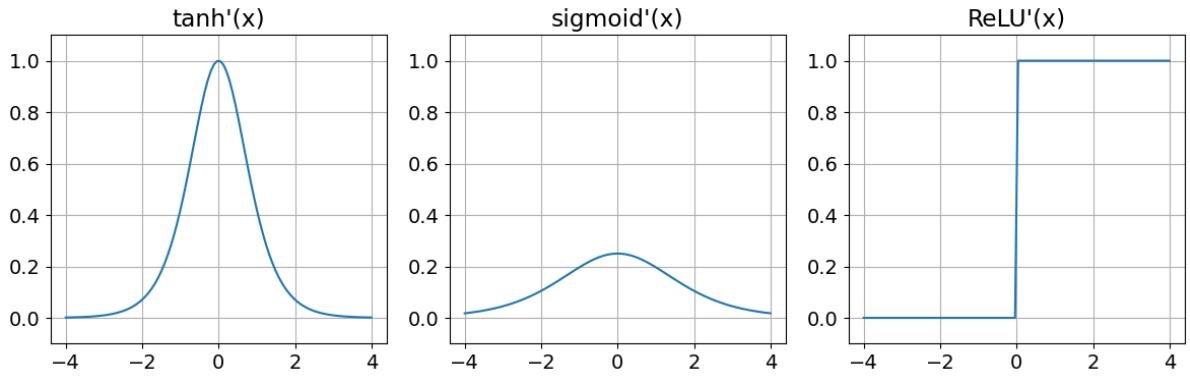
Il y a des idées importantes à saisir ici.

Tout d'abord, il faut remarquer que les poids qui sont plus éloignés de la sortie du modèle héritent de règles de gradient composées de plus de termes. Par conséquent, lorsque certains de ces termes deviennent de plus en plus petits, il y a un risque plus élevé pour ces poids que leurs gradients tombent à 0. C'est ce qu'on appelle l'effet de **gradient évanescent** (*vanishing gradient*), qui est un phénomène très courant dans les réseaux neuronaux profonds (c'est-à-dire les réseaux composés de nombreuses couches).

Deuxièmement, certains termes sont répétés dans ces formules, et en général, des termes de la forme $\frac{\partial a^{(\ell)}}{\partial o^{(\ell)}}$ et $\frac{\partial o^{(\ell)}}{\partial a^{(\ell-1)}}$ sont présents à plusieurs endroits. Ces termes peuvent être développés comme suit :

$$\frac{\partial a^{(\ell)}}{\partial o^{(\ell)}} = \varphi'(o^{(\ell)})$$
$$\frac{\partial o^{(\ell)}}{\partial a^{(\ell-1)}} = w^{(\ell-1)}$$

Voyons à quoi ressemblent les dérivées des fonctions d'activation standard :



On peut constater que la dérivée de ReLU possède une plus grande plage de valeurs d'entrée pour lesquelles elle est non nulle (typiquement toute la plage de valeurs d'entrée positives) que ses concurrentes, ce qui en fait une fonction d'activation très intéressante pour les réseaux neuronaux profonds, car nous avons vu que le terme $\frac{\partial a^{(\ell)}}{\partial o^{(\ell)}}$ apparaît de manière répétée dans les dérivations en chaîne.

4.4 Coder tout cela en `keras`

Dans `keras`, les informations sur les pertes et l'optimiseur sont transmises au moment de la compilation :

```
import keras_core as keras
from keras.layers import Dense, InputLayer
from keras.models import Sequential

model = Sequential([
    InputLayer(input_shape=(10, )),
    Dense(units=20, activation="relu"),
    Dense(units=3, activation="softmax")
])

model.summary()
```

```
Using TensorFlow backend
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 20)                220
```

(suite sur la page suivante)

(suite de la page précédente)

```
 dense_1 (Dense)             (None, 3)                 63

=================================================================
Total params: 283 (1.11 KB)
Trainable params: 283 (1.11 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model.compile(loss="categorical_crossentropy", optimizer="adam")
```

En termes de pertes :

— `"mse"` est la perte d'erreur quadratique moyenne,
— `"binary_crossentropy"` est la perte logistique pour la classification binaire,
— `"categorical_crossentropy"` est la perte logistique pour la classification multi-classes.

Les optimiseurs définis dans cette section sont disponibles sous forme de `"sgd"` et `"adam"`. Afin d'avoir le contrôle sur les hyper-paramètres des optimiseurs, on peut alternativement utiliser la syntaxe suivante :

```
from keras.optimizers import Adam, SGD

# Not a very good idea to tune beta_1
# and beta_2 parameters in Adam
adam_opt = Adam(learning_rate=0.001,
                beta_1=0.9, beta_2=0.9)

# In order to use SGD with a custom learning rate:
# sgd_opt = SGD(learning_rate=0.001)

model.compile(loss="categorical_crossentropy", optimizer=adam_opt)
```

4.5 Prétraitement des données

En pratique, pour que la phase d'ajustement du modèle se déroule correctement, il est important de mettre à l'échelle les données d'entrée. Dans l'exemple suivant, nous allons comparer deux entraînements du même modèle, avec une initialisation similaire et la seule différence entre les deux sera de savoir si les données d'entrée sont centrées-réduites ou laissées telles quelles.

```
import pandas as pd
from keras.utils import to_categorical

iris = pd.read_csv("../data/iris.csv", index_col=0)
iris = iris.sample(frac=1)
y = to_categorical(iris["target"])
X = iris.drop(columns=["target"])
```

```
from keras.layers import Dense, InputLayer
from keras.models import Sequential
from keras.utils import set_random_seed
```

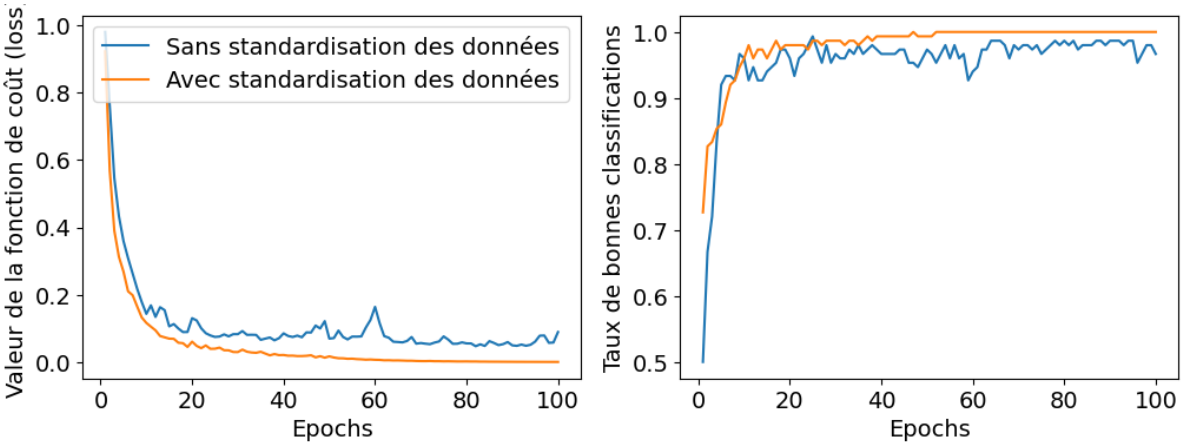
(suite sur la page suivante)

(suite de la page précédente)

```
set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y, epochs=n_epochs, batch_size=30, verbose=0)
```

Standardisons maintenant nos données et comparons les performances obtenues :

```
X -= X.mean(axis=0)
X /= X.std(axis=0)


set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h_standardized = model.fit(X, y, epochs=n_epochs, batch_size=30, verbose=0)
```



CHAPITRE 5

RÉGULARISATION

Comme nous l'avons vu dans les chapitres précédents, l'une des forces des réseaux neuronaux est qu'ils peuvent approximer n'importe quelle fonction continue lorsqu'un nombre suffisant de paramètres est utilisé. Lors de l'utilisation d'approximateurs universels dans des contextes d'apprentissage automatique, un risque connexe important est celui du surajustement (*overfitting*) aux données d'apprentissage. Plus formellement, étant donné un jeu de données d'apprentissage $\mathcal{D}_t$ tiré d'une distribution inconnue $\mathcal{D}$, les paramètres du modèle sont optimisés de manière à minimiser le risque empirique :

$$\mathcal{R}_e(\theta) = \frac{1}{|\mathcal{D}_t|} \sum_{(x_i,y_i)\in\mathcal{D}_t} \mathcal{L}(x_i, y_i; m_\theta)$$

alors que le véritable objectif est de minimiser le « vrai » risque :

$$\mathcal{R}(\theta) = \mathbb{E}_{x,y\sim\mathcal{D}}\mathcal{L}(x, y; m_\theta)$$

et les deux objectifs n'ont pas le même minimiseur.

Pour éviter cet écueil, il faut utiliser des techniques de régularisation, telles que celles présentées ci-après.

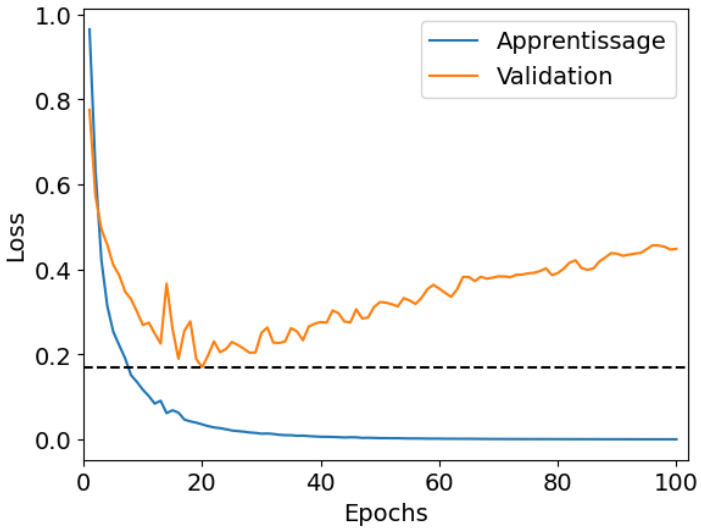
5.1 *Early stopping*

Comme illustré ci-dessous, on peut observer que l'entraînement d'un réseau neuronal pendant un trop grand nombre d'"*epochs* peut conduire à un surajustement. Notez qu'ici, le risque réel est estimé grâce à l'utilisation d'un ensemble de validation qui n'est pas vu pendant l'entraînement.

```
Using TensorFlow backend
```

```
iris = pd.read_csv("../data/iris.csv", index_col=0)
iris = iris.sample(frac=1)
y = to_categorical(iris["target"])
X = iris.drop(columns=["target"])
X -= X.mean(axis=0)
X /= X.std(axis=0)
```

```
import keras_core as keras
from keras.layers import Dense, InputLayer
from keras.models import Sequential
from keras.utils import set_random_seed


set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y, validation_split=0.3, epochs=n_epochs, batch_size=30, verbose=0)
```



Ici, le meilleur modèle (en termes de capacités de généralisation) semble être le modèle à l'*epoch* 20. En d'autres termes, si nous avions arrêté le processus d'apprentissage après l'*epoch* 20, nous aurions obtenu un meilleur modèle que si nous utilisons le modèle entraîné pendant 70 *epochs*.

C'est toute l'idée derrière la stratégie d'*early stopping*, qui consiste à arrêter le processus d'apprentissage dès que la perte de validation cesse de s'améliorer. Cependant, comme on peut le voir dans la visualisation ci-dessus, la perte de validation a tendance à osciller, et on attend souvent plusieurs *epochs* avant de supposer que la perte a peu de chances de s'améliorer dans le futur. Le nombre d'*epochs* à attendre est appelé le paramètre de *patience*.

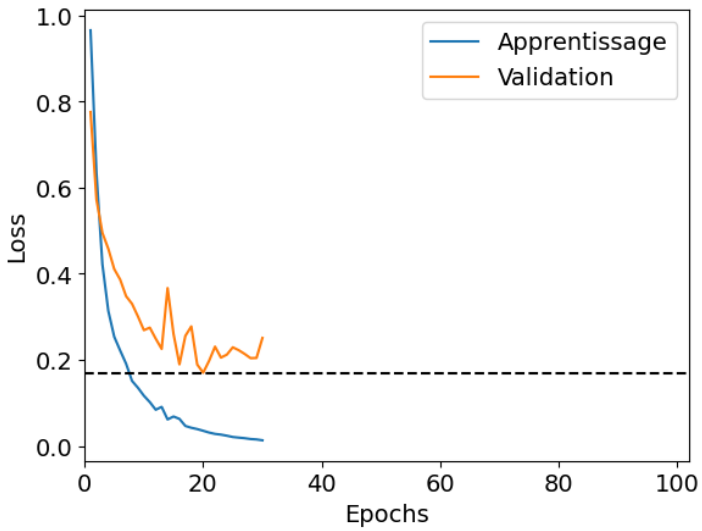
Dans `keras`, l'arrêt anticipé peut être configuré via un *callback*, comme dans l'exemple suivant :

```
from keras.callbacks import EarlyStopping


set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
```

(suite sur la page suivante)

(suite de la page précédente)

```
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=3, activation="softmax")
])

cb_es = EarlyStopping(monitor="val_loss", patience=10, restore_best_weights=True)

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y,
              validation_split=0.3, epochs=n_epochs, batch_size=30,
              verbose=0, callbacks=[cb_es])
```



Et maintenant, même si le modèle était prévu pour être entraîné pendant 70 *epochs*, l'entraînement est arrêté dès qu'il atteint 10 *epochs* consécutives sans amélioration de la perte de validation, et les paramètres du modèle sont restaurés comme les paramètres du modèle à l'"*epoch* 20.

5.2 Pénalisation de la perte

Une autre façon importante d'appliquer la régularisation dans les réseaux neuronaux est la pénalisation des pertes. Un exemple typique de cette stratégie de régularisation est la régularisation L2. Si nous désignons par $\mathcal{L}_r$ la perte régularisée par L2, elle peut être exprimée comme suit :

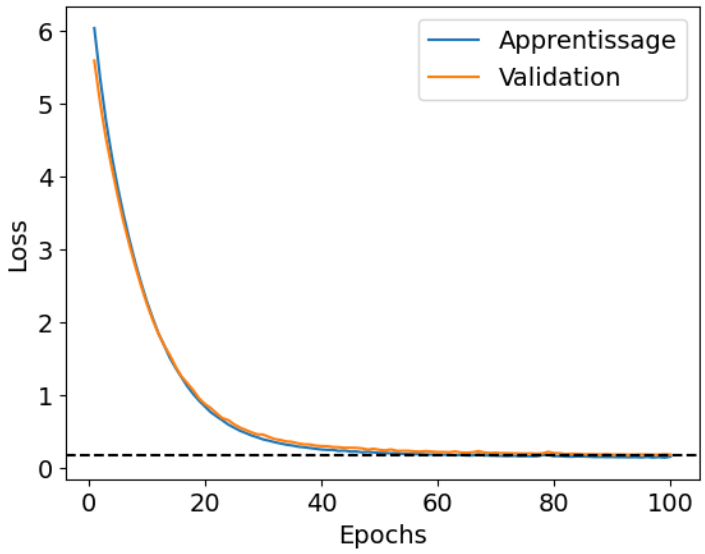
$$\mathcal{L}_r(\mathcal{D}; m_\theta) = \mathcal{L}(\mathcal{D}; m_\theta) + \lambda \sum_\ell \|\theta^{(\ell)}\|_2^2$$

où $\theta^{(\ell)}$ est la matrice de poids de la couche ℓ.

Cette régularisation tend à réduire les grandes valeurs des paramètres pendant le processus d'apprentissage, ce qui est connu pour aider à améliorer la généralisation.

En `keras`, ceci est implémenté comme :

```
from keras.regularizers import L2

λ = 0.01

set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu", kernel_regularizer=L2(λ)),
    Dense(units=256, activation="relu", kernel_regularizer=L2(λ)),
    Dense(units=256, activation="relu", kernel_regularizer=L2(λ)),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y, validation_split=0.3, epochs=n_epochs, batch_size=30, verbose=0)
```



5.3 *DropOut*

Dans cette section, nous présentons la stratégie *DropOut*, qui a été introduite dans [Srivastava *et al.*, 2014]. L'idée derrière le *DropOut* est d'éteindre certains neurones pendant l'apprentissage. Les neurones désactivés changent à chaque *minibatch* de sorte que, globalement, tous les neurones sont entraînés pendant tout le processus.

Le concept est très similaire dans l'esprit à une stratégie utilisée pour l'entraînement des forêts aléatoires, qui consiste à sélectionner aléatoirement des variables candidates pour chaque division d'arbre à l'intérieur d'une forêt, ce qui est connu pour conduire à de meilleures performances de généralisation pour les forêts aléatoires. La principale différence ici est que l'on peut non seulement désactiver les *neurones d'entrée* mais aussi les *neurones de la couche cachée* pendant l'apprentissage.

Dans `keras`, ceci est implémenté comme une couche, qui agit en désactivant les neurones de la couche précédente dans le réseau :

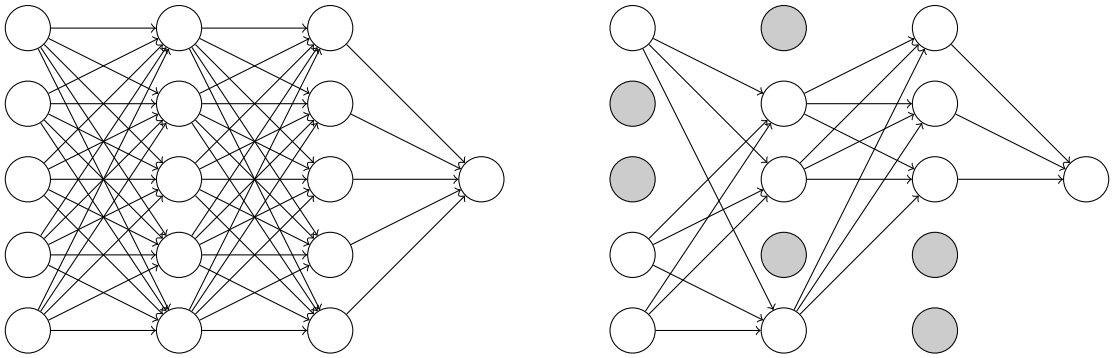
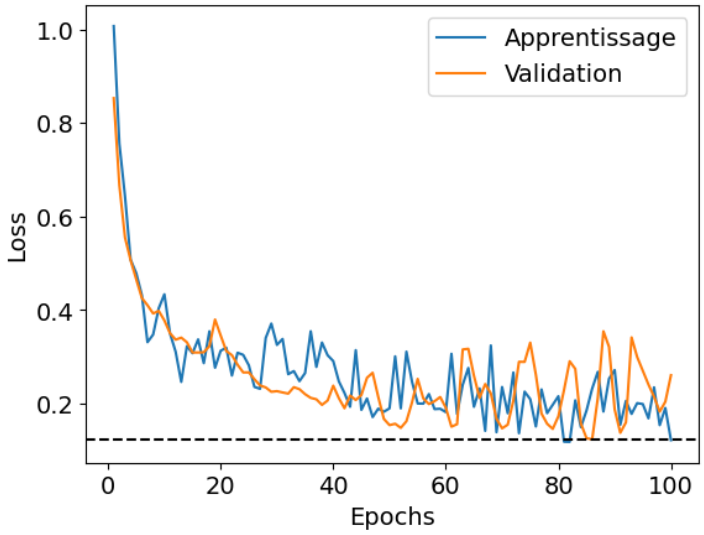


FIG. 5.1 – Illustration du mécanisme de *DropOut*. Afin d'entraîner un modèle donné (à gauche), à chaque *minibatch*, une proportion donnée de neurones est choisie au hasard pour être « désactivée » et le sous-réseau résultant est utilisé pour l'étape d'optimisation en cours (*cf.* figure de droite, dans laquelle 40% des neurones – colorés en gris – sont désactivés).

```
from keras.layers import Dropout

set_random_seed(0)
switchoff_proba = 0.3
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dropout(rate=switchoff_proba),
    Dense(units=256, activation="relu"),
    Dropout(rate=switchoff_proba),
    Dense(units=256, activation="relu"),
    Dropout(rate=switchoff_proba),
    Dense(units=256, activation="relu"),
    Dropout(rate=switchoff_proba),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y, validation_split=0.3, epochs=n_epochs, batch_size=30, verbose=0)
```



Exercice #1

En observant les valeurs de perte dans la figure ci-dessus, pouvez-vous expliquer pourquoi la perte de validation est presque systématiquement inférieure à celle calculée sur le jeu d'apprentissage ?

Solution

En fait, la perte d'apprentissage est calculée comme la perte moyenne sur tous les *minibatchs* d'apprentissage pendant une *epoch*. Si nous nous rappelons que pendant l'apprentissage, à chaque *minibatch*, 30% des neurones sont désactivés, on peut voir que seule une sous-partie du modèle complet est utilisée lors de l'évaluation de la perte d'apprentissage alors que le modèle complet est utilisé lors de la prédiction sur l'ensemble de validation, ce qui explique pourquoi la perte de validation mesurée est inférieure à celle de l'apprentissage.

CHAPITRE 6

RÉSEAUX NEURONAUX CONVOLUTIFS

Les réseaux de neurones convolutifs (aussi appelés ConvNets) sont conçus pour tirer parti de la structure des données. Dans ce chapitre, nous aborderons deux types de réseaux convolutifs : nous commencerons par le cas monodimensionnel et verrons comment les réseaux convolutifs à convolutions 1D peuvent être utiles pour traiter les séries temporelles. Nous présenterons ensuite le cas 2D, particulièrement utile pour traiter les données d'image.

6.1 Réseaux de neurones convolutifs pour les séries temporelles

Les réseaux de neurones convolutifs pour les séries temporelles reposent sur l'opérateur de convolution 1D qui, étant donné une série temporelle $\mathbf{x}$ et un filtre $\mathbf{f}$, calcule une carte d'activation comme :

$$(\mathbf{x} * \mathbf{f})(t) = \sum_{k=-L}^{L} f_k x_{t+k} \tag{6.1}$$

où le filtre $\mathbf{f}$ est de longueur $(2L + 1)$.

Le code suivant illustre cette notion en utilisant un filtre gaussien :

Les réseaux de neurones convolutifs sont constitués de blocs de convolution dont les paramètres sont les coefficients des filtres qu'ils intègrent (les filtres ne sont donc pas fixés *a priori* comme dans l'exemple ci-dessus mais plutôt appris). Ces blocs de convolution sont équivariants par translation, ce qui signifie qu'un décalage (temporel) de leur entrée entraîne le même décalage temporel de leur sortie :

```
/tmp/ipykernel_14676/368849627.py:32: UserWarning: This figure includes Axes that␣
↪are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
```

```
<IPython.core.display.HTML object>
```

Les modèles convolutifs sont connus pour être très performants dans les applications de vision par ordinateur, utilisant des quantités modérées de paramètres par rapport aux modèles entièrement connectés (bien sûr, des contre-exemples existent, et le terme « modéré » est particulièrement vague).

La plupart des architectures standard de séries temporelles qui reposent sur des blocs convolutionnels sont des adaptations directes de modèles de la communauté de la vision par ordinateur ([Le Guennec *et al.*, 2016] s'appuie sur une alternance entre couches de convolution et couches de *pooling*, tandis que des travaux plus récents s'appuient sur des connexions résiduelles et des modules d'*inception* [Fawaz *et al.*, 2020]). Ces blocs de base (convolution, pooling, couches résiduelles) sont discutés plus en détail dans la section suivante.

Ces modèles de classification des séries temporelles (et bien d'autres) sont présentés et évalués dans [Fawaz *et al.*, 2019] que nous conseillons au lecteur intéressé.

6.2 Réseaux de neurones convolutifs pour les images

Nous allons maintenant nous intéresser au cas 2D, dans lequel les filtres de convolution ne glisseront pas sur un seul axe comme dans le cas des séries temporelles, mais plutôt sur les deux dimensions (largeur et hauteur) d'une image.

6.2.1 Images et convolutions

Comme on le voit ci-dessous, une image est une grille de pixels, et chaque pixel a une valeur d'intensité dans chacun des canaux de l'image. Les images couleur sont typiquement composées de 3 canaux (ici Rouge, Vert et Bleu).

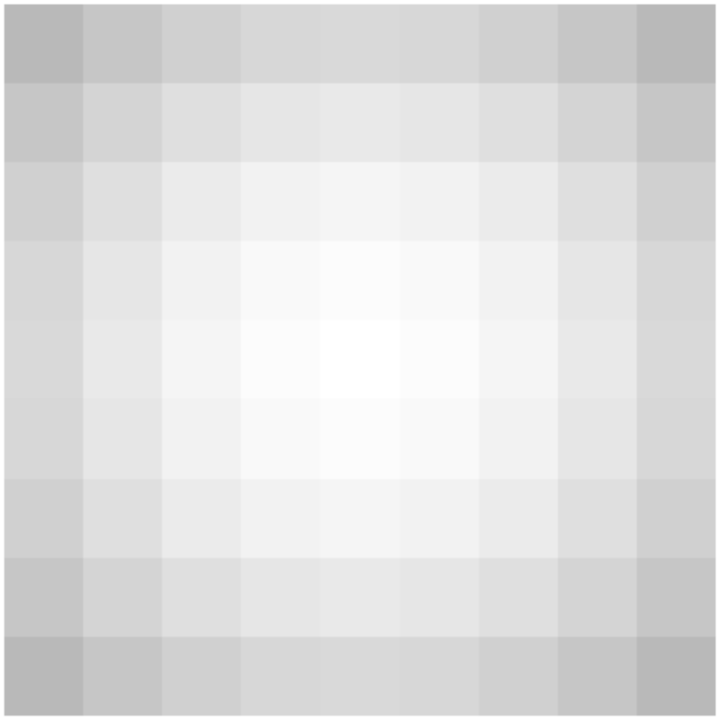
FIG. 6.1 – Une image et ses 3 canaux (intensités de Rouge, Vert et Bleu, de gauche à droite).

La sortie d'une convolution sur une image $\mathbf{x}$ est une nouvelle image, dont les valeurs des pixels peuvent être calculées comme suit :

$$(\mathbf{x} * \mathbf{f})(i,j) = \sum_{k=-K}^{K} \sum_{l=-L}^{L} \sum_{c=1}^{3} f_{k,l,c} x_{i+k,j+l,c}. \tag{6.2}$$

En d'autres termes, les pixels de l'image de sortie sont calculés comme le produit scalaire entre un filtre de convolution (qui est un tenseur de forme $(2K+1, 2L+1, c)$) et un *patch* d'image centré à la position donnée.

Considérons, par exemple, le filtre de convolution 9x9 suivant :



Le résultat de la convolution de l'image de chat ci-dessus avec ce filtre est l'image suivante en niveaux de gris (c'est-à-dire constituée d'un seul canal) :

On peut remarquer que cette image est une version floue de l'image originale. C'est parce que nous avons utilisé un filtre Gaussien. Comme pour les séries temporelles, lors de l'utilisation d'opérations de convolution dans les réseaux neuronaux, le contenu des filtres sera appris, plutôt que défini *a priori*.

6.2.2 Réseaux convolutifs de type LeNet

Dans [LeCun *et al.*, 1998], un empilement de couches de convolution, de *pooling* et de couches entièrement connectées est introduit pour une tâche de classification d'images, plus spécifiquement une application de reconnaissance de chiffres. Le réseau neuronal résultant, appelé LeNet, est représenté ci-dessous :

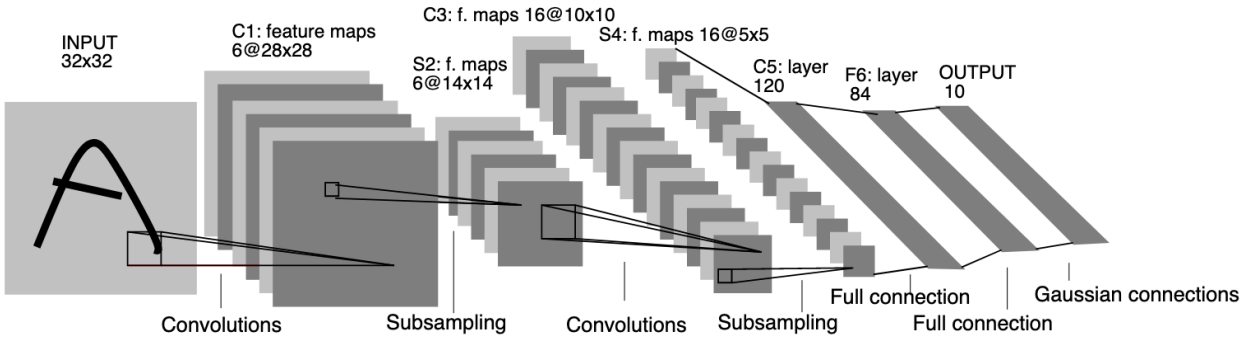


FIG. 6.2 – Modèle LeNet-5

Couches de convolution

Une couche de convolution est constituée de plusieurs filtres de convolution (également appelés *kernels*) qui opèrent en parallèle sur la même image d'entrée. Chaque filtre de convolution génère une carte d'activation en sortie et toutes ces cartes sont empilées pour former la sortie de la couche de convolution. Tous les filtres d'une couche partagent la même largeur et la même hauteur. Un terme de biais et une fonction d'activation peuvent être utilisés dans les couches de convolution, comme dans d'autres couches de réseaux neuronaux. Dans l'ensemble, la sortie d'un filtre de convolution est calculée comme suit :

$$(\mathbf{x} * \mathbf{f})(i,j,c) = \varphi\left(\sum_{k=-K}^{K}\sum_{l=-L}^{L}\sum_{c'} f^{c}_{k,l,c'} x_{i+k,j+l,c'} + b_c\right) \tag{6.3}$$

où c désigne le canal de sortie (notez que chaque canal de sortie est associé à un filtre f^c), b_c est le terme de biais qui lui est associé et φ est la fonction d'activation utilisée.

Astuce: En `keras`, une telle couche est implémentée à l'aide de la classe `Conv2D` :

```
import keras_core as keras
from keras.layers import Conv2D

layer = Conv2D(filters=6, kernel_size=5, padding="valid", activation="relu")
```

Padding

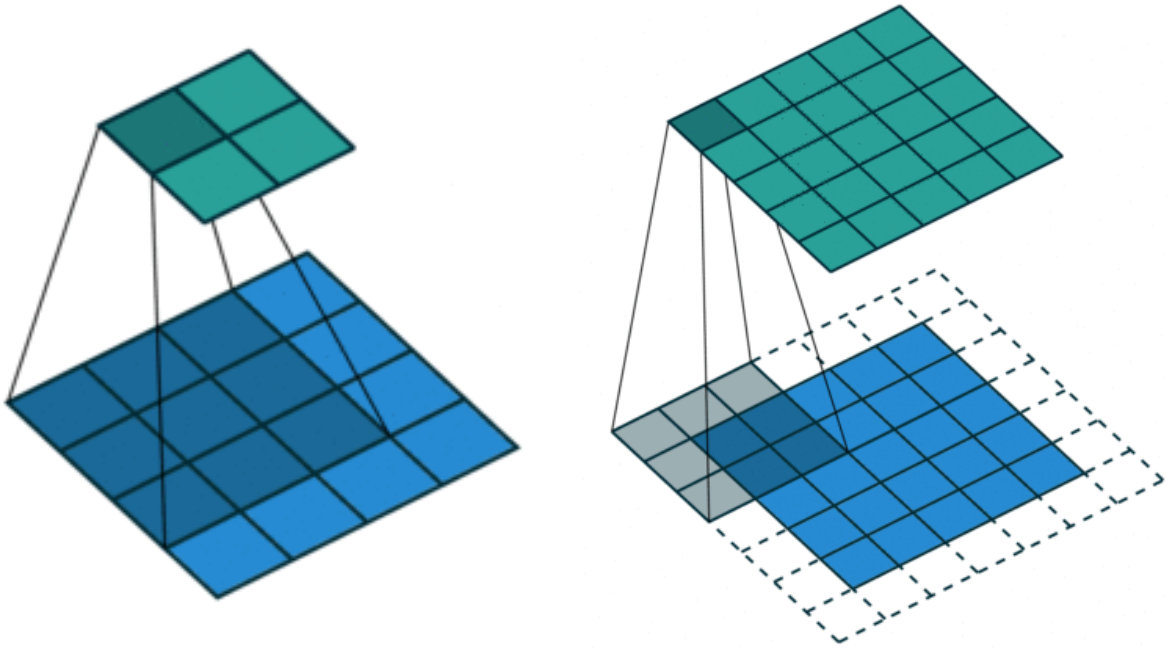


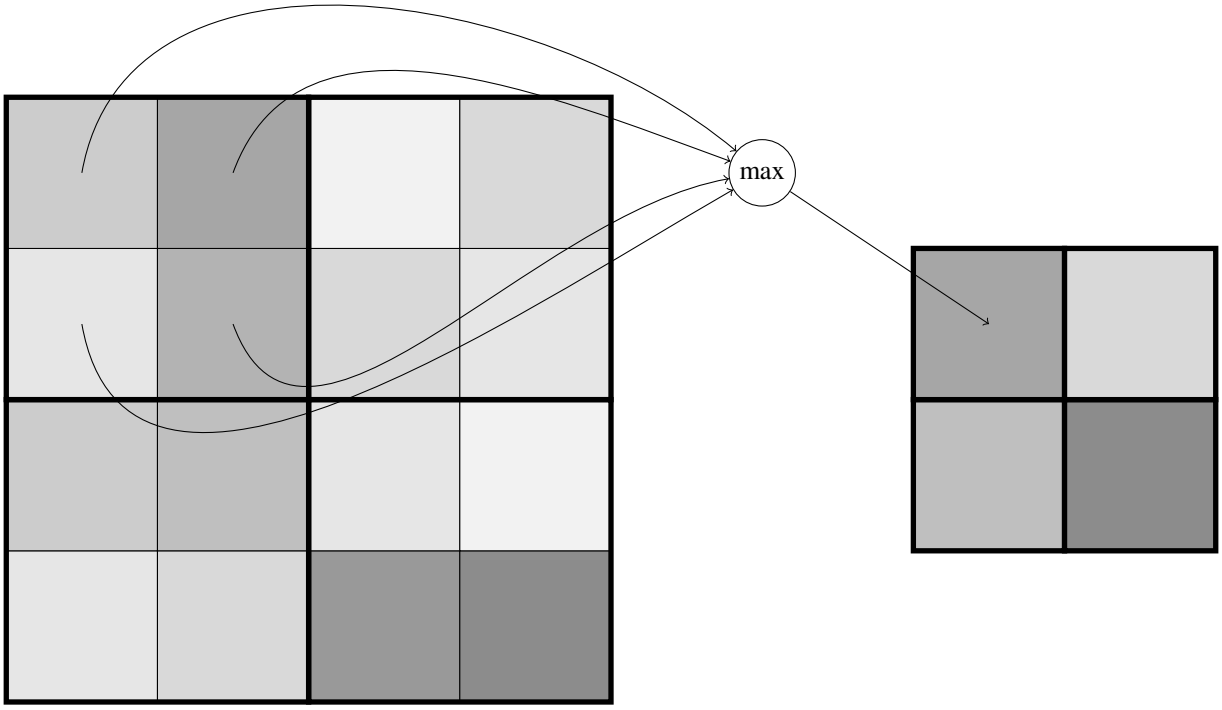
FIG. 6.3 – Visualisation de l'effet du *padding* (source: V. Dumoulin, F. Visin - A guide to convolution arithmetic for deep learning). Gauche: sans *padding*, droite: avec *padding*.

Lors du traitement d'une image d'entrée, il peut être utile de s'assurer que la carte de caractéristiques (ou carte d'activation) de sortie a la même largeur et la même hauteur que l'image d'entrée. Cela peut être réalisé en agrandissant artificiellement l'image d'entrée et en remplissant les zones ajoutées avec des zéros, comme illustré dans Fig. 6.3 dans lequel la zone de *padding* est représentée en blanc.

Couches de *pooling*

Les couches de *pooling* effectuent une opération de sous-échantillonnage qui résume en quelque sorte les informations contenues dans les cartes de caractéristiques dans des cartes à plus faible résolution.

L'idée est de calculer, pour chaque parcelle d'image, une caractéristique de sortie qui calcule un agrégat des pixels de la parcelle. Les opérateurs d'agrégation typiques sont les opérateurs de moyenne (dans ce cas, la couche correspondante est appelée *average pooling*) ou de maximum (pour les couches de *max pooling*). Afin de réduire la résolution des cartes de sortie, ces agrégats sont généralement calculés sur des fenêtres glissantes qui ne se chevauchent pas, comme illustré ci-dessous, pour un *max pooling* avec une taille de *pooling* de 2x2 :



Ces couches étaient largement utilisées historiquement dans les premiers modèles convolutifs et le sont de moins en moins à mesure que la puissance de calcul disponible augmente.

Astuce: En `keras`, les couches de *pooling* sont implémentées à travers les classes `MaxPool2D` et `AvgPool2D` :

```
from keras.layers import MaxPool2D, AvgPool2D

max_pooling_layer = MaxPool2D(pool_size=2)
average_pooling_layer = AvgPool2D(pool_size=2)
```

Ajout d'une *tête de classification*

Un empilement de couches de convolution et de *pooling* produit une carte d'activation structurée (qui prend la forme d'une grille 2d avec une dimension supplémentaire pour les différents canaux). Lorsque l'on vise une tâche de classification d'images, l'objectif est de produire la classe la plus probable pour l'image d'entrée, ce qui est généralement réalisé par une tête de classification (*classification head*) composée de couches entièrement connectées.

Pour que la tête de classification soit capable de traiter une carte d'activation, les informations de cette carte doivent être transformées en un vecteur. Cette opération est appelée *Flatten* dans `keras`, et le modèle correspondant à Fig. 6.2 peut être implémenté comme :

```
from keras.models import Sequential
from keras.layers import InputLayer, Conv2D, MaxPool2D, Flatten, Dense

model = Sequential([
```

(suite sur la page suivante)

(suite de la page précédente)

```
    InputLayer(input_shape=(32, 32, 1)),
    Conv2D(filters=6, kernel_size=5, padding="valid", activation="relu"),
    MaxPool2D(pool_size=2),
    Conv2D(filters=16, kernel_size=5, padding="valid", activation="relu"),
    MaxPool2D(pool_size=2),
    Flatten(),
    Dense(120, activation="relu"),
    Dense(84, activation="relu"),
    Dense(10, activation="softmax")
])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         156

 max_pooling2d (MaxPooling2  (None, 14, 14, 6)         0
 D)

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        2416

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 16)          0
 g2D)

 flatten (Flatten)           (None, 400)               0

 dense (Dense)               (None, 120)               48120

 dense_1 (Dense)             (None, 84)                10164

 dense_2 (Dense)             (None, 10)                850

=================================================================
Total params: 61706 (241.04 KB)
Trainable params: 61706 (241.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

CHAPITRE 7

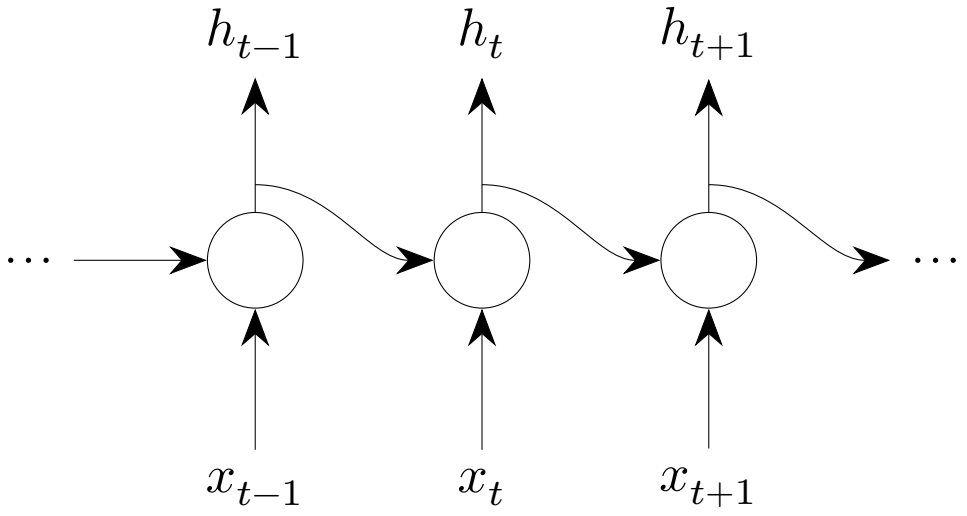
RÉSEAUX NEURONAUX RÉCURRENTS

Les réseaux neuronaux récurrents (RNN) traitent les éléments d'une série temporelle un par un. Typiquement, à l'instant t, un bloc récurrent prend en entrée :

— l'entrée courante x_t et

— un état caché h_{t-1} qui a pour but de résumer les informations clés provenant de des entrées passées $\{x_0, \dots, x_{t-1}\}$

Ce bloc retourne un état caché mis à jour h_t :



Il existe différentes couches récurrentes qui diffèrent principalement par la façon dont h_t est calculée.

7.1 Réseaux récurrents standard

La formulation originale d'une RNN est la suivante :

$$\forall t, h_t = \tanh(W_h h_{t-1} + W_x x_t + b) \tag{7.1}$$

où W_h est une matrice de poids associée au traitement de l'état caché précédent, W_x est une autre matrice de poids associée au traitement de la l'entrée actuelle et b est un terme de biais.

On notera ici que W_h, W_x et b ne sont pas indexés par t, ce qui signifie que qu'ils sont **partagés entre tous les temps**.

Une limitation importante de cette formule est qu'elle échoue à capturer les dépendances à long terme. Pour mieux comprendre pourquoi, il faut se rappeler que les paramètres de ces réseaux sont optimisés par des algorithmes de descente de gradient stochastique.

Pour simplifier les notations, considérons un cas simplifié dans lequel h_t et x_t sont tous deux des valeurs scalaires, et regardons ce que vaut le gradient de la sortie h_t par rapport à W_h (qui est alors aussi un scalaire) :

$$\nabla_{W_h}(h_t) = \tanh'(o_t) \cdot \frac{\partial o_t}{\partial W_h} \tag{7.2}$$

où $o_t = W_h h_{t-1} + W_x x_t + b$, donc:

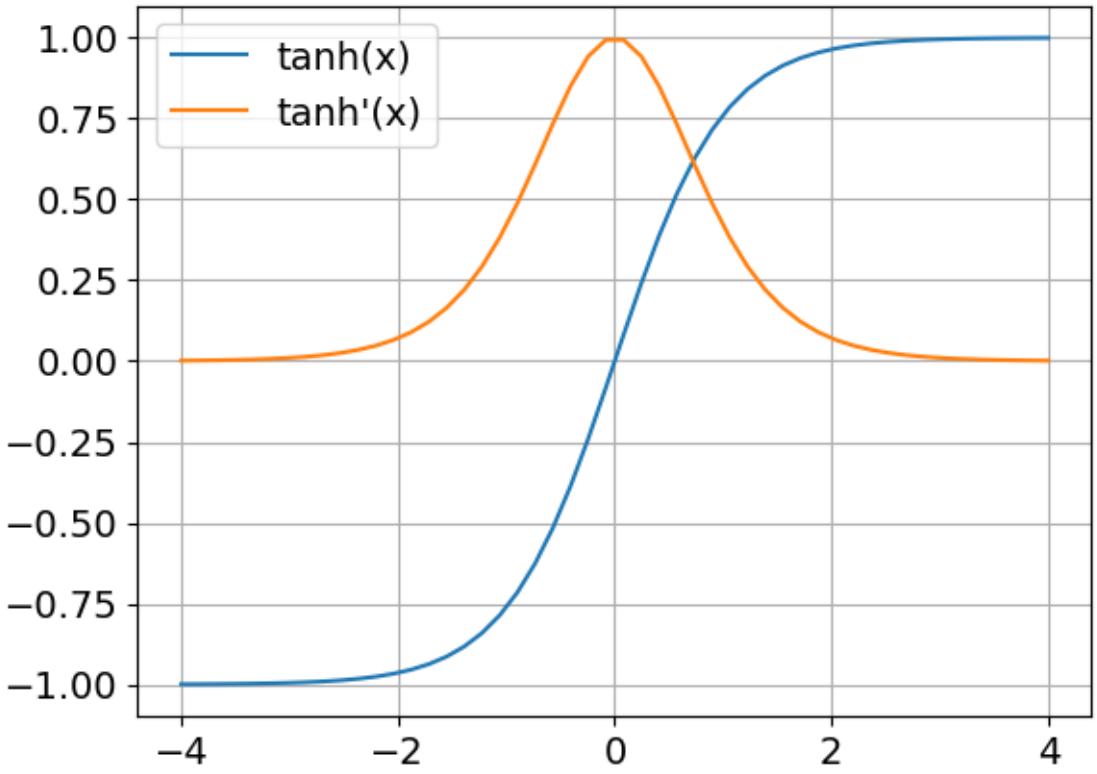
$$\frac{\partial o_t}{\partial W_h} = h_{t-1} + W_h \cdot \frac{\partial h_{t-1}}{\partial W_h}\,. \tag{7.3}$$

Ici, la forme de $\frac{\partial h_{t-1}}{\partial W_h}$ sera similaire à celle de $\nabla_{W_h}(h_t)$ ci-dessus, et, au final, on obtient :

$$\begin{aligned}
\nabla_{W_h}(h_t) &= \tanh'(o_t) \cdot \left[h_{t-1} + W_h \cdot \frac{\partial h_{t-1}}{\partial W_h}\right] && (7.4)\\
&= \tanh'(o_t) \cdot \left[h_{t-1} + W_h \cdot \tanh'(o_{t-1}) \cdot [h_{t-2} + W_h \cdot [...]]\right] && (7.5)\\
&= h_{t-1}\tanh'(o_t) + h_{t-2} W_h \tanh'(o_t)\tanh'(o_{t-1}) + ... && (7.6)\\
&= \sum_{t'=1}^{t-1} h_{t'} \left[W_h^{t-t'-1}\tanh'(o_{t'+1}) \cdot \cdots \cdot \tanh'(o_t)\right] && (7.7)
\end{aligned}$$

En d'autres termes, l'influence de $h_{t'}$ sera atténuée par un facteur $W_h^{t-t'-1}\tanh'(o_{t'+1}) \cdot \cdots \cdot \tanh'(o_t)$.

Rappelons maintenant à quoi ressemblent la fonction tanh et sa dérivée :



On peut voir à quel point les gradients se rapprochent rapidement de 0 pour des entrées plus grandes (en valeur absolue) que 2, et avoir plusieurs termes de ce type dans une dérivation en chaîne fera tendre les termes correspondants vers 0.

En d'autres termes, le gradient de l'état caché au temps t sera seulement influencé par quelques uns de ses prédécesseurs $\{h_{t-1}, h_{t-2}, ...\}$ et les les dépendances à long terme seront ignorées lors de l'actualisation des paramètres du modèle par descente de gradient. Il s'agit d'une occurrence d'un phénomène plus général connu sous le nom de *vanishing gradient*.

7.2 Long Short Term Memory

Les blocs *Long Short Term Memory* (LSTM, [Hochreiter and Schmidhuber, 1997]) ont été conçus comme une alternative à aux blocs récurrents classiques. Ils visent à atténuer l'effet de *vanishing gradient* par l'utilisation de portes qui codent explicitement quelle partie de l'information doit (resp. ne doit pas) être utilisée.

Les portes dans les réseaux neuronaux

Dans la terminologie des réseaux de neurones, une porte $g \in [0,1]^d$ est un vecteur utilisé pour filtrer les informations d'un vecteur caractéristique entrant $v \in \mathbb{R}^d$ de telle sorte que le résultat de l'application de la porte est : $g \odot v$. où $\odot$ est le produit élément-par-élément. La porte g aura donc tendance à supprimer une partie des caractéristiques de v. (celles qui correspondent à des valeurs très faibles de g).

Dans ces blocs, un état supplémentaire est utilisé, appelé état de la cellule C_t. Cet état est calculé comme suit :

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{7.8}$$

où f_t est appelée *forget gate* (elle pousse le réseau à oublier les parties inutiles de l'état passé de la cellule), i_t est l'"*input gate* et $\tilde{C}_t$ est une version actualisée de l'état de la cellule (qui, à son tour, peut être partiellement censurée par l'"*input gate*).

Laissons de côté pour l'instant les détails concernant le calcul de ces 3 termes et concentrons-nous plutôt sur la façon dont la formule ci-dessus est est significativement différente de la règle de mise à jour de l'état caché dans le modèle classique. En effet, dans ce cas, si le réseau l'apprend (par l'intermédiaire de f_t), l'information complète de l'état précédent de la cellule C_{t-1} peut être récupérée, ce qui permet aux gradients de se propager à rebours de l'axe du temps (et de ne plus disparaître).

Alors, le lien entre l'état de la cellule et l'état caché est :

$$h_t = o_t \odot \tanh(C_t)\,. \tag{7.9}$$

En d'autres termes, l'état caché est la version transformée (par la fonction tanh) de l'état de la cellule, encore censuré par une porte de sortie (*output gate*) o_t.

Toutes les portes utilisées dans les formules ci-dessus sont définies de manière similaire :

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{7.10}$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{7.11}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{7.12}$$

où σ est la fonction d'activation sigmoïde (dont les valeurs sont comprises dans $[0, 1]$) et $[h_{t-1}, x_t]$ la concaténation des caractéristiques h_{t-1} et x_t.

Enfin, l'état de cellule mis à jour $\tilde{C_t}$ est calculé comme suit :

$$\tilde{C_t} = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)\,. \tag{7.13}$$

Il existe dans la littérature de nombreuses variantes de ces blocs LSTM qui reposent toujours sur les mêmes principes de base.

7.3 Gated Recurrent Unit

Une paramétrisation légèrement différente d'un bloc récurrent est utilisée dans les Gated Recurrent Units (GRU, [Cho *et al.*, 2014]).

Les GRUs reposent également sur l'utilisation de portes pour laisser (de manière adaptative) l'information circuler à travers le temps. Une première différence significative entre les GRUs et les LSTMs est que les GRUs n'ont pas recours à l'utilisation d'un état de cellule. Au lieu de cela, la règle de mise à jour de l'état caché est la suivante :

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{7.14}$$

où z_t est une porte qui équilibre (par caractéristique) la quantité d'informations qui est conservée de l'état caché précédent avec la quantité d'informations qui doit être mise à jour en utilisant le nouvel état caché candidat $\tilde{h}_t$, calculé comme suit :

$$\tilde{h}_t = \tanh(W \cdot [r_t \odot h_{t-1}, x_t] + b)\,, \tag{7.15}$$

où r_t est une porte supplémentaire qui peut cacher une partie de l'état caché précédent.

Les formules pour les portes z_t et r_t sont similaires à celles fournies pour f_t, i_t et o_t dans le cas des LSTMs.

Une étude graphique de la capacité de ces variantes de réseaux récurrents à apprendre des dépendances à long terme est fournie dans [Madsen, 2019].

7.4 Conclusion

Dans ce chapitre et le précédent, nous avons passé en revue les architectures de réseaux de neurones qui sont utilisées pour apprendre à partir de données temporelles ou séquentielles. En raison de contraintes de temps, nous n'avons pas abordé les modèles basés sur l'attention dans ce cours. Nous avons présenté les modèles convolutifs qui visent à extraire des formes locales discriminantes dans les séries et les modèles récurrents qui exploitent plutôt la notion de séquence. Concernant ces derniers, des variantes visant à faire face à l'effet de gradient évanescent ont été introduites. Il est à noter que les modèles récurrents sont connus pour nécessiter plus de données d'entraînement que leurs homologues convolutifs.

BIBLIOGRAPHIE

[Goh17] Gabriel Goh. Why momentum really works. *Distill*, 2017. URL: http://distill.pub/2017/momentum.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: a method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *ICLR*. 2015.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56) :1929–1958, 2014. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[FFW+19] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, 33(4) :917–963, 2019.

[FLF+20] Hassan Ismail Fawaz, Benjamin Lucas, Germain Forestier, Charlotte Pelletier, Daniel F Schmidt, Jonathan Weber, Geoffrey I Webb, Lhassane Idoumghar, Pierre-Alain Muller, and François Petitjean. Inceptiontime: finding alexnet for time series classification. *Data Mining and Knowledge Discovery*, 34(6) :1936–1962, 2020.

[LGMT16] Arthur Le Guennec, Simon Malinowski, and Romain Tavenard. Data Augmentation for Time Series Classification using Convolutional Neural Networks. In *ECML/PKDD Workshop on Advanced Analytics and Learning on Temporal Data*. Riva Del Garda, Italy, September 2016.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11) :2278–2324, 1998.

[CVMerrienboerBB14] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: encoder-decoder approaches. 2014. arXiv:1409.1259.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8) :1735–1780, 1997.

[Mad19] Andreas Madsen. Visualizing memorization in rnns. *Distill*, 2019. URL: https://distill.pub/2019/memorization-in-rnns.