Introduction au Deep Learning (notes de cours)

Romain Tavenard

févr. 07, 2023

CONTENTS

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 sont en cours de traduction vers le français

CHAPTER 1

INTRODUCTION

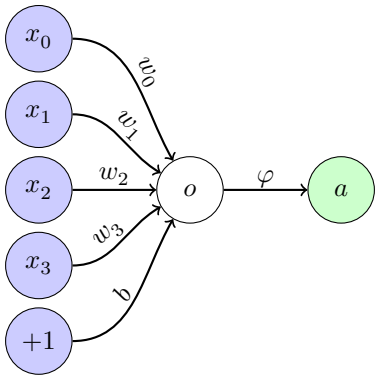
In this introduction chapter, we will present a first neural network called the Perceptron. This model is a neural network made of a single neuron, and we will use it here as a way to introduce key concepts that we will detail later in the course.

1.1 A first model: the Perceptron

In the neural network terminology, a neuron is a parametrized function that takes a vector **x** as input and outputs a single value a as follows:

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

where the parameters of the neuron are its weights stored in **w** and a bias term b, and φ is an activation function that is chosen *a priori* (we will come back to it in more details later in the course):



A model made of a single neuron is called a Perceptron.

1.2 Optimization

The models presented in this book are aimed at solving prediction problems, in which the goal is to find « good enough » parameter values for the model at stake given some observed data.

The problem of finding such parameter values is coined optimization and the deep learning field makes extensive use of a specific family of optimization strategies called **gradient descent**.

1.2.1 Gradient Descent

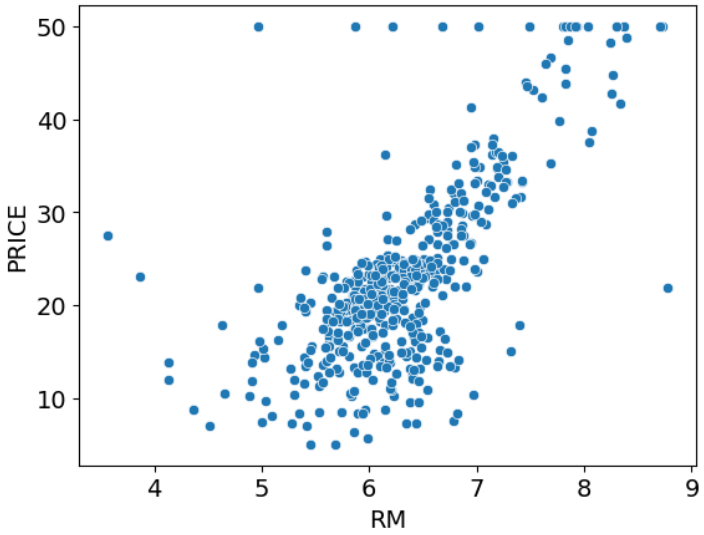
To make one's mind about gradient descent, let us assume we are given the following dataset about house prices:

```
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]
```

In our case, we will try (for a start) to predict the target value of this dataset, which is the median value of owner-occupied homes in $1000 `"PRICE"`, as a function of the average number of rooms per dwelling `"RM"` :

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



A short note on this model

In the Perceptron terminology, this model:

- has no activation function (*i.e.* φ is the identity function)
- has no bias (*i.e.* b is forced to be 0, it is not learnt)

Let us assume we have a naive approach in which our prediction model is linear without intercept, that is, for a given input x_i the predicted output is computed as:

$$\hat{y_i} = wx_i$$

where w is the only parameter of our model.

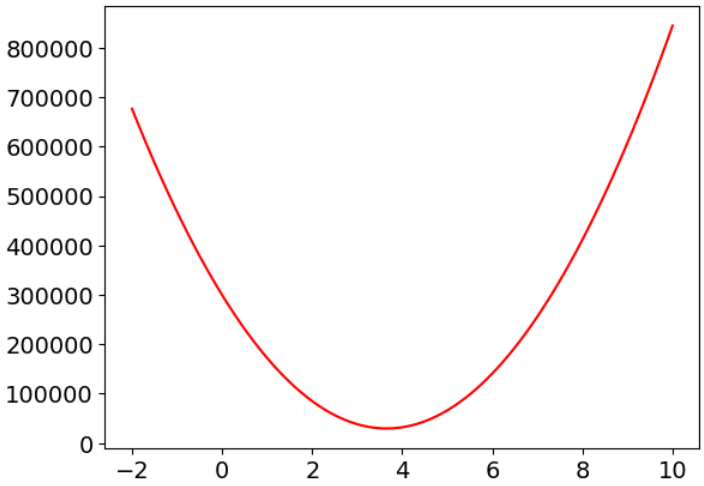
Let us further assume that the quantity we aim at minimizing (our objective, also called loss) is:

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

where y_i is the ground truth value associated with the i-th sample in our dataset.

Let us have a look at this quantity as a function of 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-");
```



Here, it seems that a value of w around 4 should be a good pick, but this method (generating lots of values for the parameter and computing the loss for each value) cannot scale to models that have lots of parameters, so we will try something else.

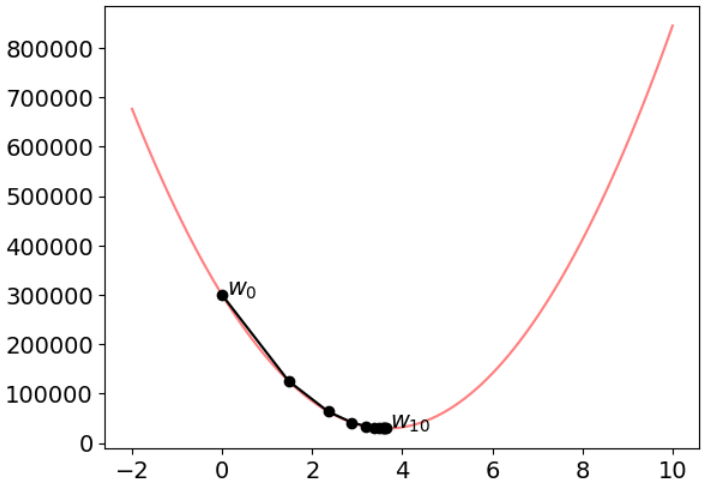
Let us suppose we have access, each time we pick a candidate value for w, to both the loss $\mathcal{L}$ and information about how $\mathcal{L}$ varies, locally. We could, in this case, compute a new candidate value for w by moving from the previous candidate value in the direction of steepest descent. This is the basic idea behind the gradient descent algorithm that, from an initial candidate w_0, iteratively computes new candidates as:

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

where ρ is a hyper-parameter (called the learning rate) that controls the size of the steps to be done, and $\left. \frac{\partial \mathcal{L}}{\partial w} \right|_{w=w_t}$ is the gradient of $\mathcal{L}$ with respect to w, evaluated at $w = w_t$. As you can see, the direction of steepest descent is the opposite of the direction pointed by the gradient (and this holds when dealing with vector parameters too).

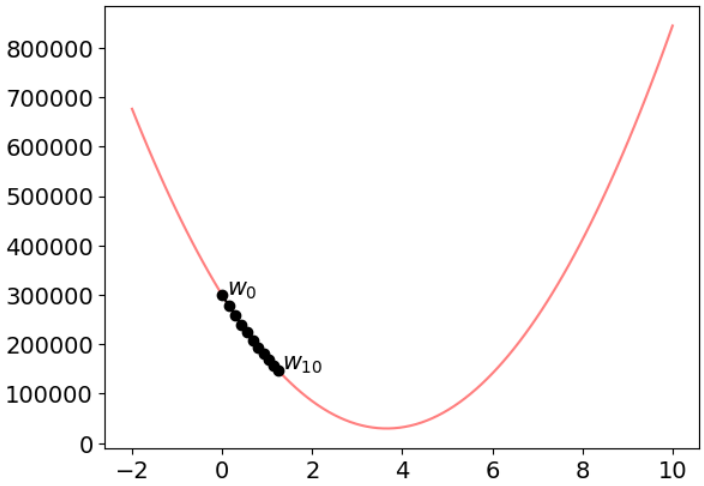
This process is repeated until convergence, as illustrated in the following visualization:

```
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="$w_{10}$");
```



What would we get if we used a smaller learning rate?

```
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="$w_{10}$");
```



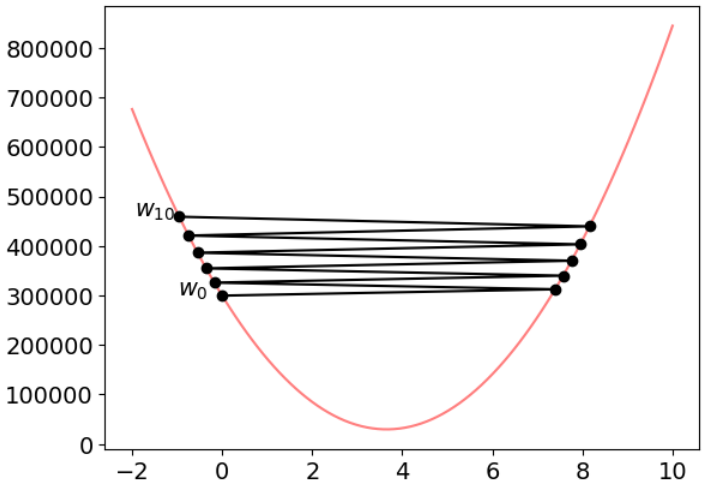
It would definitely take more time to converge. But, take care, a larger learning rate is not always a good idea:

```
rho = 5e-5

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

(suite sur la page suivante)

(suite de la page précédente)

```
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="$w_{10}$");
```



See how we are slowly diverging because our steps are too large?

1.3 Wrap-up

In this section, we have introduced:

- a very simple model, called the Perceptron: this will be a building block for the more advanced models we will detail later in the course, such as:
  - the *Multi-Layer Perceptron*
  - *Convolutional architectures*
  - *Recurrent architectures*
- the fact that a task comes with a loss function to be minimized (here, we have used the *mean squared error (MSE)* for our regression task), which will be discussed in *a dedicated chapter*;
- the concept of gradient descent to optimize the chosen loss over a model's single parameter, and this will be extended in *our chapter on optimization*.

CHAPTER 2

PERCEPTRONS MULTICOUCHES

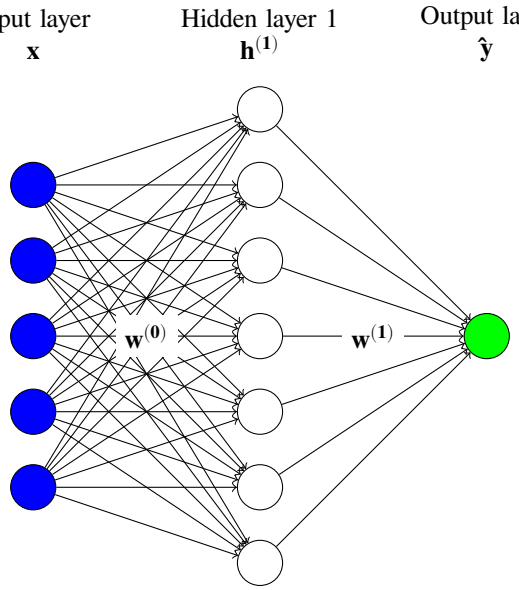
Dans le chapitre précédent, nous avons vu un modèle très simple appelé le perceptron. Dans ce modèle, la sortie prédite $\hat{y}$ est calculée comme une combinaison linéaire des caractéristiques d'entrée plus un biais :

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

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 :



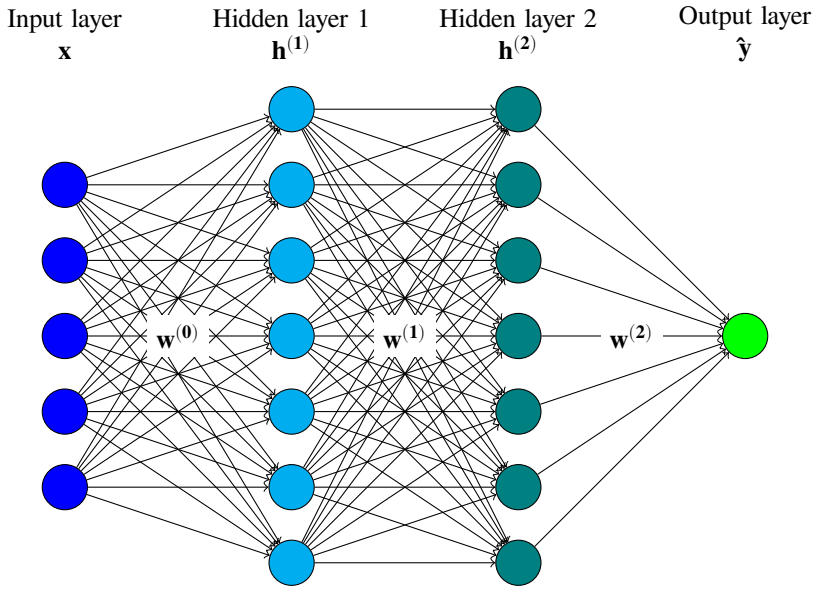
La question que l'on peut se poser maintenant est de savoir si cette couche cachée supplémentaire permet effectivement de couvrir une plus grande famille de modèles. C'est à cela que sert le théorème d'approximation universelle ci-dessous.

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.

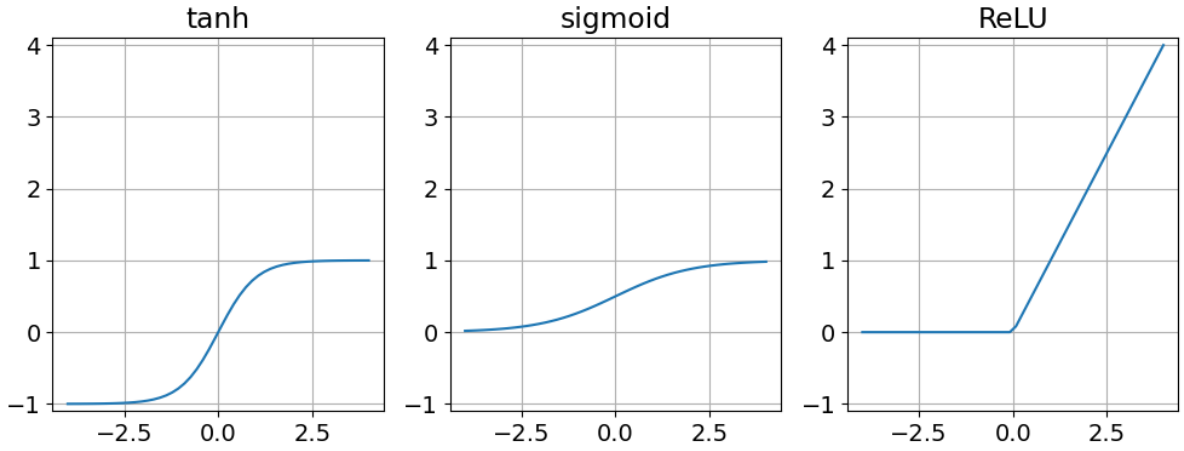
Il est important de noter que si nous utilisons la fonction identité comme fonction d'activation, quelle que soit la profondeur de notre MLP, nous ne couvrirons plus que la famille des modèles linéaires. En pratique, nous utiliserons donc des fonctions d'activation qui ont un certain régime linéaire 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 :

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

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

$$\text{ReLU}(x) = \begin{cases} x \text{ if } x > 0 \\ 0 \text{ otherwise} \end{cases}$$



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 :

```
from tensorflow.keras.layers import Dense, InputLayer
from tensorflow.keras.models import Sequential

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

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
```

(suite sur la page suivante)

(suite de la page précédente)

```
 dense (Dense)                (None, 20)                220

 dense_1 (Dense)              (None, 3)                 63

=================================================================
Total params: 283
Trainable params: 283
Non-trainable params: 0
_________________________________________________________________
```

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
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]
```

CHAPTER 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 :

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

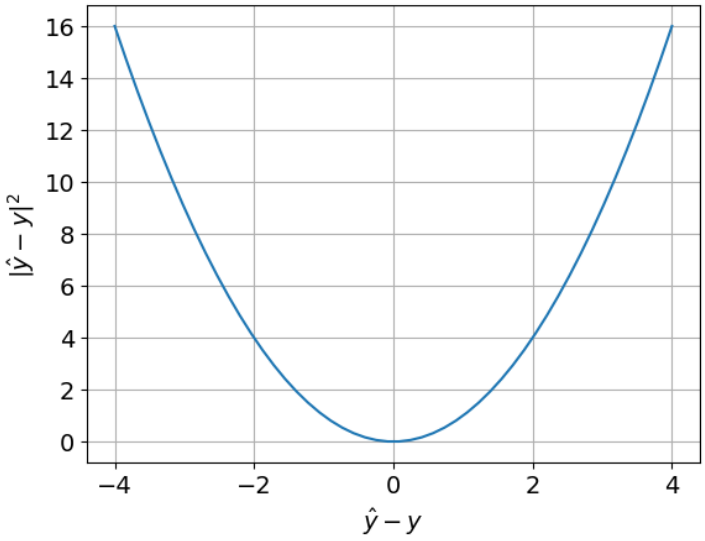
où m_θ est notre modèle et θ est l'ensemble de tous ses paramètres (poids et biais).

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 :



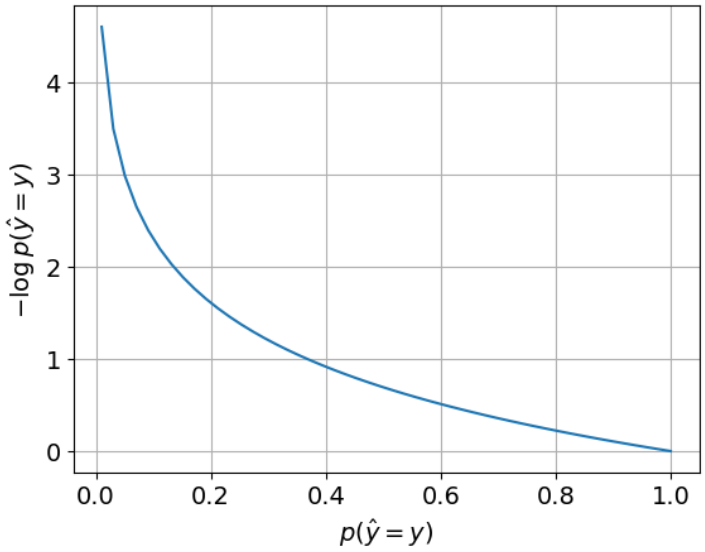
3.2 Perte logistique

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 :



CHAPTER 4

OPTIMIZATION

In this chapter, we will present variants of the **Gradient Descent** optimization strategy and show how they can be used to optimize neural network parameters.

Let us start with the basic Gradient Descent algorithm and its limitations.

Algorithm 1 (Gradient Descent)

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

1. Initialize model parameters θ
2. for $e = 1..E$
    1. for $(x_i, y_i) \in \mathcal{D}$
        1. Compute prediction $\hat{y}_i = m_\theta(x_i)$
        2. Compute gradient $\nabla_\theta \mathcal{L}_i$
    2. Compute overall gradient $\nabla_\theta \mathcal{L} = \frac{1}{n} \sum_i \nabla_\theta \mathcal{L}_i$
    3. Update parameters θ based on $\nabla_\theta \mathcal{L}$

The typical update rule for the parameters θ at iteration t is

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

where ρ is an important hyper-parameter of the method, called the learning rate. Basically, gradient descent updates θ in the direction of steepest decrease of the loss $\mathcal{L}$.

As one can see in the previous algorithm, when performing gradient descent, model parameters are updated once per epoch, which means a full pass over the whole dataset is required before the update can occur. When dealing with large datasets, this is a strong limitation, which motivates the use of stochastic variants.

4.1 Stochastic Gradient Descent (SGD)

The idea behind the Stochastic Gradient Descent algorithm is to get cheap estimates for the quantity

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

where $\mathcal{D}$ is the whole training set. To do so, one draws subsets of data, called *minibatches*, and

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

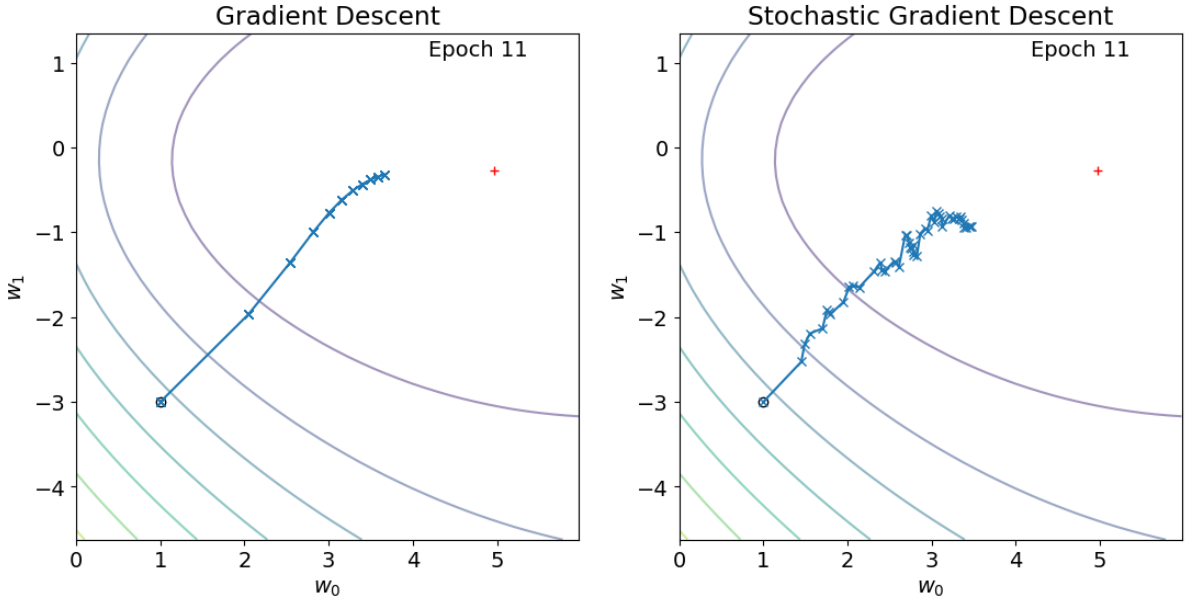
is used as an estimator for $\nabla_\theta \mathcal{L}(\mathcal{D}; m_\theta)$. This results in the following algorithm in which, interestingly, parameter updates occur after each minibatch, which is multiple times per epoch.

Algorithm 2 (Stochastic Gradient Descent)

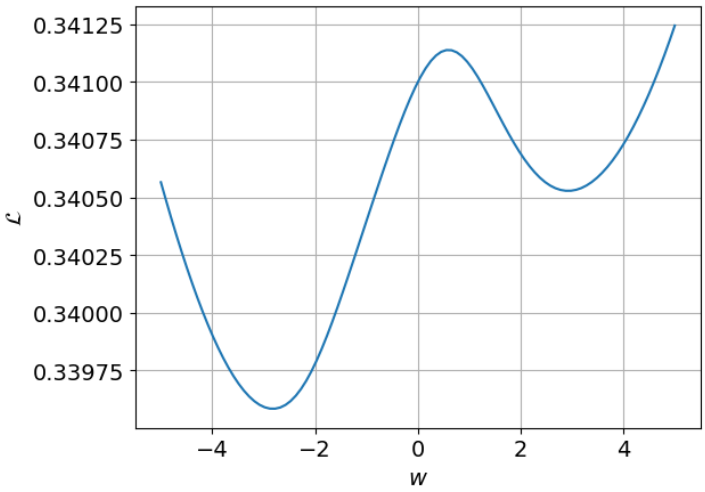
Input: A dataset $\mathcal{D} = (X, y)$

1. Initialize model parameters θ
2. for $e = 1..E$
    1. for $t = 1..n_{\text{minibatches}}$
        1. Draw minibatch $\mathcal{B}$ as a random sample of size b from $\mathcal{D}$
        2. for $(x_i, y_i) \in \mathcal{B}$
            1. Compute prediction $\hat{y}_i = m_\theta(x_i)$
            2. Compute gradient $\nabla_\theta \mathcal{L}_i$
        3. Compute minibatch-level gradient $\nabla_\theta \mathcal{L}_\mathcal{B} = \frac{1}{b} \sum_i \nabla_\theta \mathcal{L}_i$
        4. Update parameters θ based on $\nabla_\theta \mathcal{L}_\mathcal{B}$

As a consequence, when using SGD, parameter updates are more frequent, but they are « noisy » since they are based on an minibatch estimation of the gradient instead of relying on the true gradient, as illustrated below:



Apart from implying more frequent parameter updates, SGD has an extra benefit in terms of optimization, which is key for neural networks. Indeed, as one can see below, contrary to what we had in the Perceptron case, the MSE loss (and the same applies for the logistic loss) is no longer convex in the model parameters as soon as the model has at least one hidden layer:



Gradient Descent is known to suffer from local optima, and such loss landscapes are a serious problem for GD. On the other hand, Stochastic Gradient Descent is likely to benefit from noisy gradient estimations to escape local minima.

4.2 A note on Adam

Adam [Kingma and Ba, 2015] is a variant of the Stochastic Gradient Descent method. It differs in the definition of the steps to be performed at each parameter update.

First, it uses what is called momentum, which basically consists in relying on past gradient updates to smooth out the trajectory in parameter space during optimization. An interactive illustration of momentum can be found in [Goh, 2017].

The resulting plugin replacement for the gradient is:

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

When β_1 is zero, we have $\mathbf{m}^{(t+1)} = \nabla_\theta \mathcal{L}$ and for $\beta_1 \in]0,1[$, $\mathbf{m}^{(t+1)}$ balances the current gradient estimate with information about past estimates, stored in $\mathbf{m}^{(t)}$.

Another important difference between SGD and the Adam variant consists in using an adaptive learning rate. In other words, instead of using the same learning rate ρ for all model parameters, the learning rate for a given parameter θ_i is defined as:

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

where ϵ is a small constant and

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

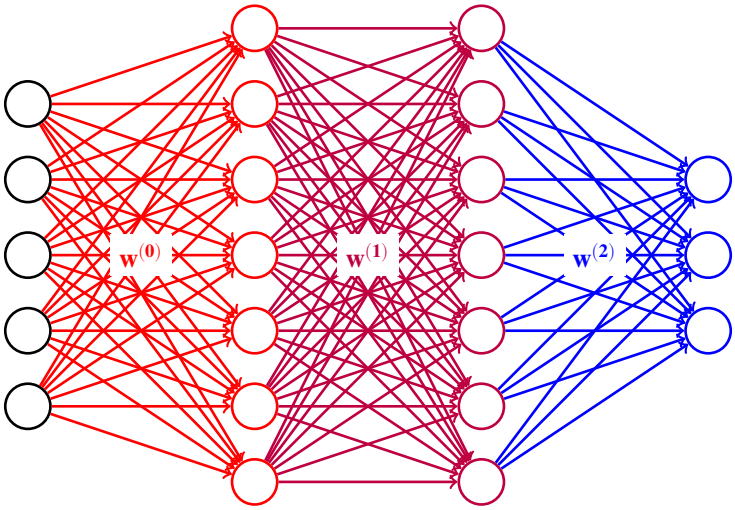
Here also, the s term uses momentum. As a result, the learning rate will be lowered for parameters which have suffered large updates in the past iterations.

Overall, the Adam update rule is:

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

4.3 The curse of depth

Let us consider the following neural network:



and let us recall that, at a given layer (ℓ), the layer output is computed as

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

where φ is the activation function for the given layer (we ignore the bias terms in this simplified example).

In order to perform (stochastic) gradient descent, gradients of the loss with respect to model parameters need to be computed.

By using the chain rule, these gradients can be expressed as:

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

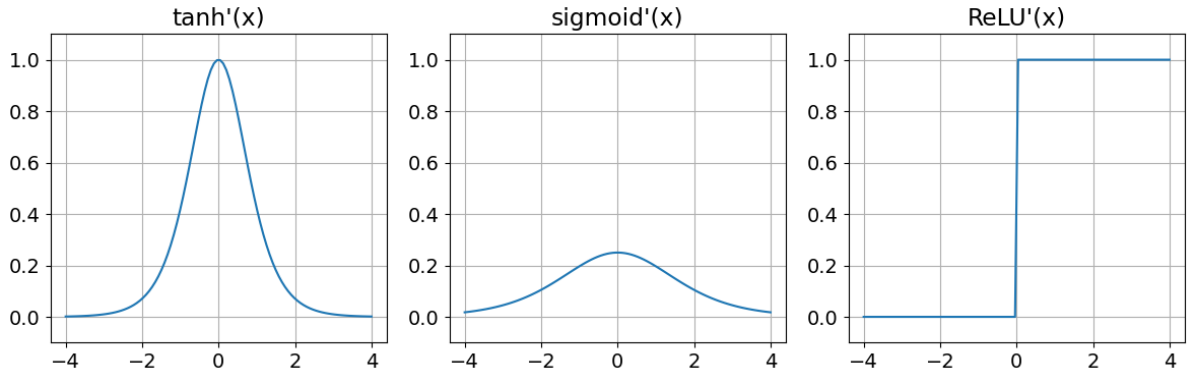
There are important insights to grasp here.

First, one should notice that weights that are further from the output of the model inherit gradient rules made of more terms. As a consequence, when some of these terms get smaller and smaller, there is a higher risk for those weights that their gradients collapse to 0, this is called the **vanishing gradient** effect, which is a very common phenomenon in deep neural networks (*i.e.* those networks made of many layers).

Second, some terms are repeated in these formulas, and in general, terms of the form $\frac{\partial a^{(\ell)}}{\partial o^{(\ell)}}$ and $\frac{\partial o^{(\ell)}}{\partial a^{(\ell-1)}}$ are present in several places. These terms can be further developed as:

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

Let us inspect what the derivatives of standard activation functions look like:



One can see that the derivative of ReLU has a wider range of input values for which it is non-zero (typically the whole range of positive input values) than its competitors, which makes it a very attractive candidate activation function for deep neural networks, as we have seen that the $\frac{\partial a^{(\ell)}}{\partial o^{(\ell)}}$ term appears repeatedly in chain rule derivations.

4.4 Wrapping things up in `keras`

In `keras`, loss and optimizer information are passed at compile time:

```
from tensorflow.keras.layers import Dense, InputLayer
from tensorflow.keras.models import Sequential

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

model.summary()
```

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

 dense_1 (Dense)              (None, 3)                 63

=================================================================
Total params: 283
Trainable params: 283
Non-trainable params: 0
_________________________________________________________________
```

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

In terms of losses:

- `"mse"` is the mean squared error loss,
- `"binary_crossentropy"` is the logistic loss for binary classification,
- `"categorical_crossentropy"` is the logistic loss for multi-class classification.

The optimizers defined in this section are available as `"sgd"` and `"adam"`. In order to get control over optimizer hyper-parameters, one can alternatively use the following syntax:

```
from tensorflow.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 Data preprocessing

In practice, for the model fitting phase to behave well, it is important to scale the input features. In the following example, we will compare two trainings of the same model, with similar initialization and the only difference between both will be whether input data is center-reduced or left as-is.

```
import pandas as pd
from tensorflow.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 tensorflow.keras.layers import Dense, InputLayer
from tensorflow.keras.models import Sequential
from tensorflow.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, epochs=n_epochs, batch_size=30, verbose=0)
```

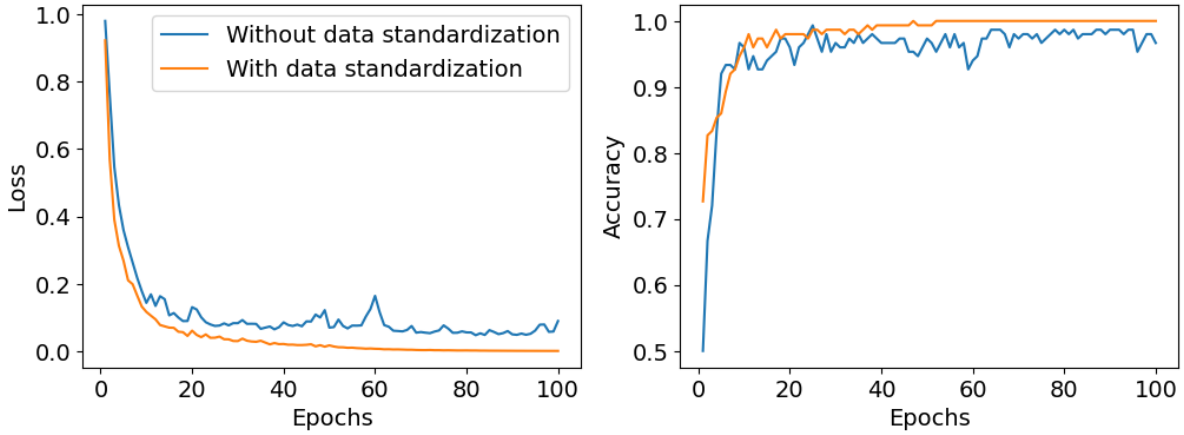
Let us now standardize our data and compare performance:

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

(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_standardized = model.fit(X, y, epochs=n_epochs, batch_size=30, verbose=0)
```



CHAPTER 5

REGULARIZATION

As discussed in previous chapters, one of the strengths of the neural networks is that they can approximate any continuous functions when a sufficient number of parameters is used. When using universal approximators in machine learning settings, an important related risk is that of overfitting the training data. More formally, given a training dataset $\mathcal{D}_t$ drawn from an unknown distribution $\mathcal{D}$, model parameters are optimized so as to minimize the empirical risk:

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

whereas the real objective is to minimize the « true » risk:

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

and both objectives do not have the same minimizer.

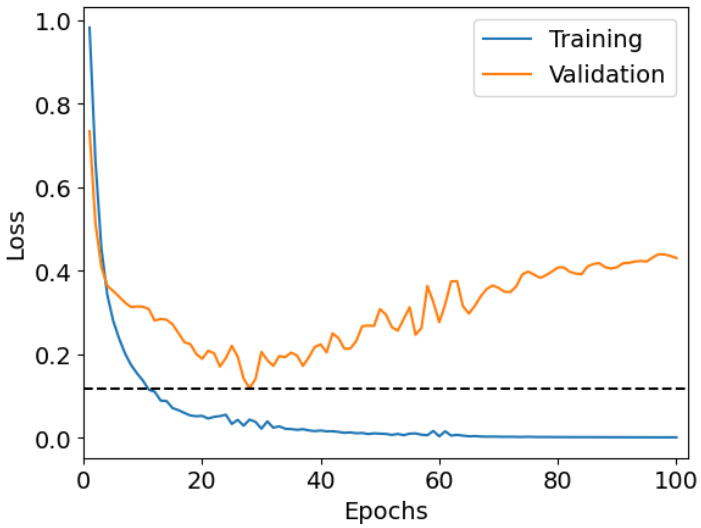
To avoid this pitfall, one should use regularization techniques, such as the ones presented in the following.

5.1 Early Stopping

As illustrated below, it can be observed that training a neural network for a too large number of epochs can lead to overfitting. Note that here, the true risk is estimated through the use of a validation set that is not seen during training.

```
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)
```

```
from tensorflow.keras.layers import Dense, InputLayer
from tensorflow.keras.models import Sequential
from tensorflow.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)
```



Here, the best model (in terms of generalization capabilities) seems to be the model at epoch `28`. In other words, if we had stopped the learning process after epoch `28`, we would have gotten a better model than if we use the model trained during 70 epochs.

This is the whole idea behind the « early stopping » strategy, which consists in stopping the learning process as soon as the validation loss stops improving. As can be seen in the visualization above, however, the validation loss tends to oscillate, and one often waits for several epochs before assuming that the loss is unlikely to improve in the future. The number of epochs to wait is called the *patience* parameter.

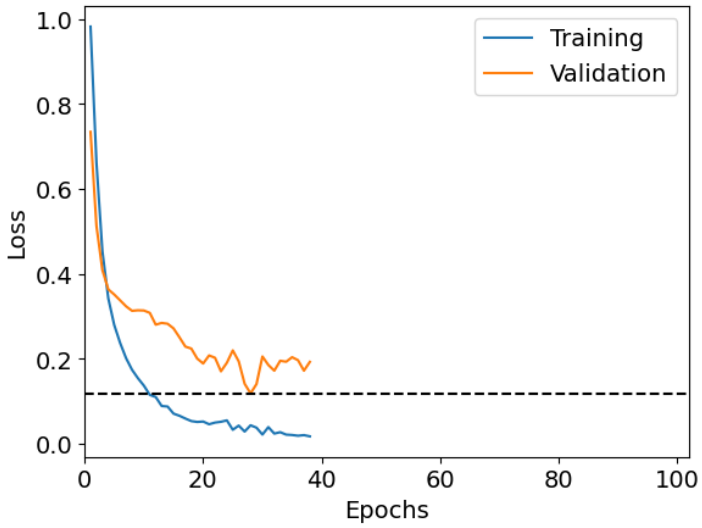
In `keras`, early stopping can be set up via a callback, as in the following example:

```
from tensorflow.keras.callbacks import EarlyStopping


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

(suite sur la page suivante)

(suite de la page précédente)

```
    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])
```



And now, even is the model was scheduled to be trained for 70 epochs, training is stopped as soon as it reaches 10 consecutive epochs without improving on the validation loss, and the model parameters are restored as the parameters of the model at epoch `28`.

5.2 Loss penalization

Another important way to enforce regularization in neural networks is through loss penalization. A typical instance of this regularization strategy is the L2 regularization. If we denote by $\mathcal{L}_r$ the L2-regularized loss, it can be expressed as:

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

where $\theta^{(\ell)}$ is the weight matrix of layer ℓ.

This regularization tends to shrink large parameter values during the learning process, which is known to help improve generalization.

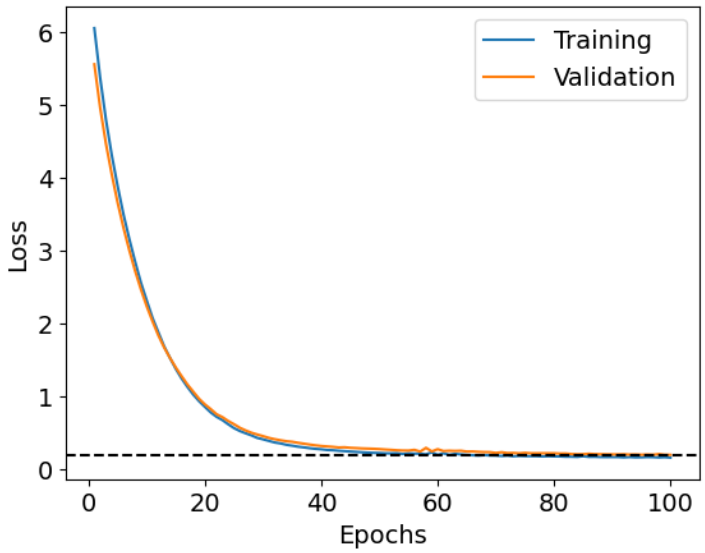
In `keras`, this is implemented as:

```
from tensorflow.keras.regularizers import L2

λ = 0.01
```

(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", 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

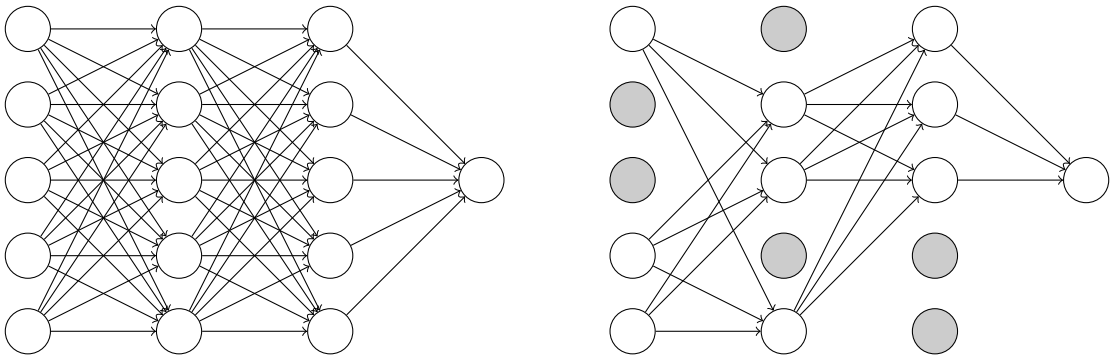


Figure5.1: Illustration of the DropOut mechanism. In order to train a given model (left), at each mini-batch, a given proportion of neurons is picked at random to be « switched off » and the subsequent sub-network is used for the current optimization step (*cf.* right-hand side figure, in which 40% of the neurons – coloured in gray – are switched off).

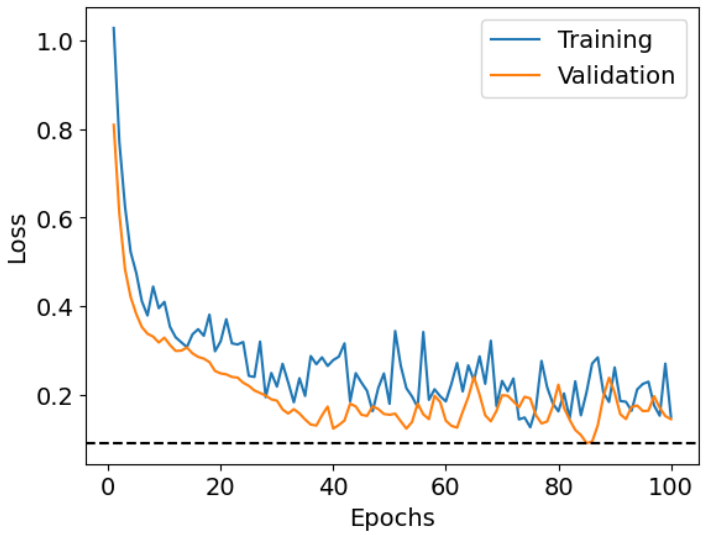
In this section, we present the DropOut strategy, which was introduced in [Srivastava *et al.*, 2014]. The idea behind

DropOut is to *switch off* some of the neurons during training. The switched off neurons change at each mini-batch such that, overall, all neurons are trained during the whole process.

The concept is very similar in spirit to a strategy that is used for training random forest, which consists in randomly selecting candidate variables for each tree split inside a forest, which is known to lead to better generalization performance for random forests. The main difference here is that one can not only switch off *input neurons* but also *hidden-layer ones* during training.

In `keras`, this is implemented as a layer, which acts by switching off neurons from the previous layer in the network:

```
from tensorflow.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)
```



Exercise #1

When observing the loss values in the figure above, can you explain why the validation loss is almost consistently lower than the training one?

Solution

In fact, the training loss is computed as the average loss over all training mini-batches during an epoch. Now, if we recall

that during training, at each minibatch, 30% of the neurons are switched-off, one can see that only a subpart of the full model is used when evaluating the training loss while the full model is retrieved when predicting on the validation set, which explains why the measured validation loss is lower than the training one.

CHAPTER 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_11081/368849627.py:32: UserWarning: This figure includes Axes that␣
↪are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
/tmp/ipykernel_11081/368849627.py:23: MatplotlibDeprecationWarning: Auto-removal␣
↪of overlapping axes is deprecated since 3.6 and will be removed two minor␣
↪releases later; explicitly call ax.remove() as needed.
  fig2 = plt.subplot(2, 1, 2)
```

```
<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).

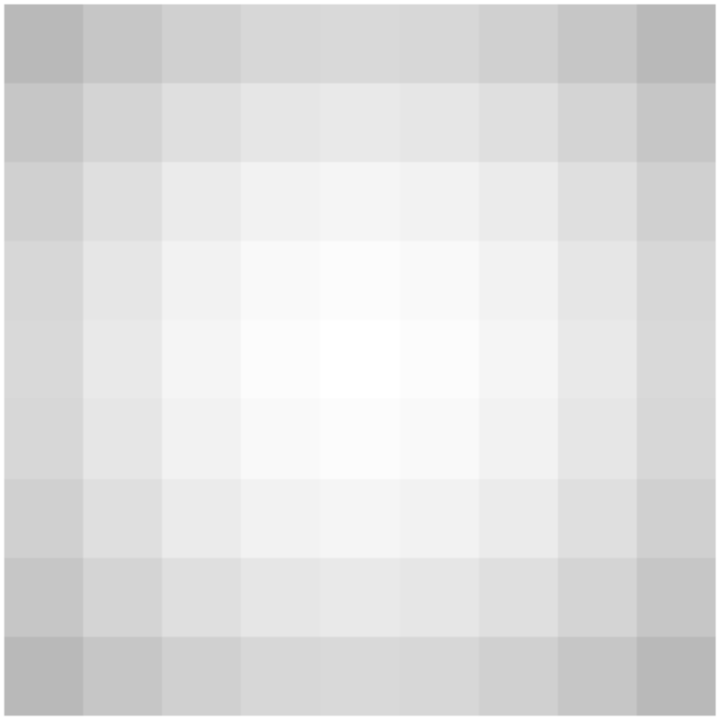
Figure6.1: Une image et ses 3 canaux (intensités de Rouge, Vert et Bleu, de gauche à droite).

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 :

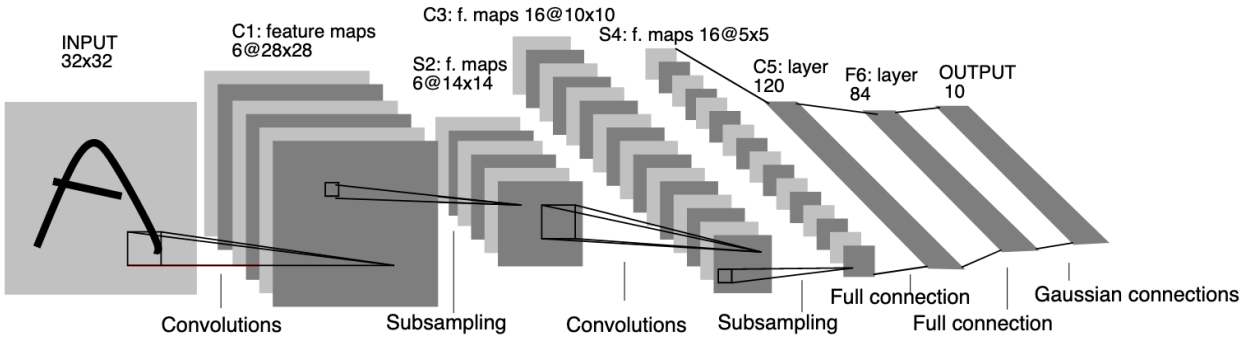


Figure6.2: Modèle LeNet-5

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` :

```
from tensorflow.keras.layers import Conv2D

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

Padding

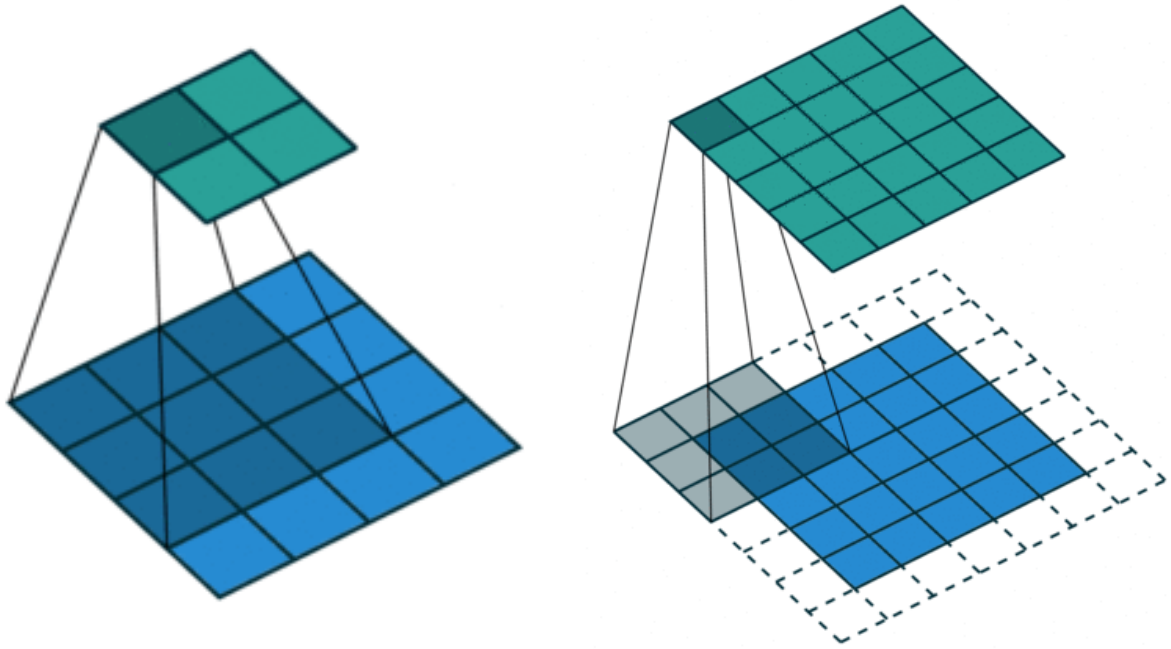


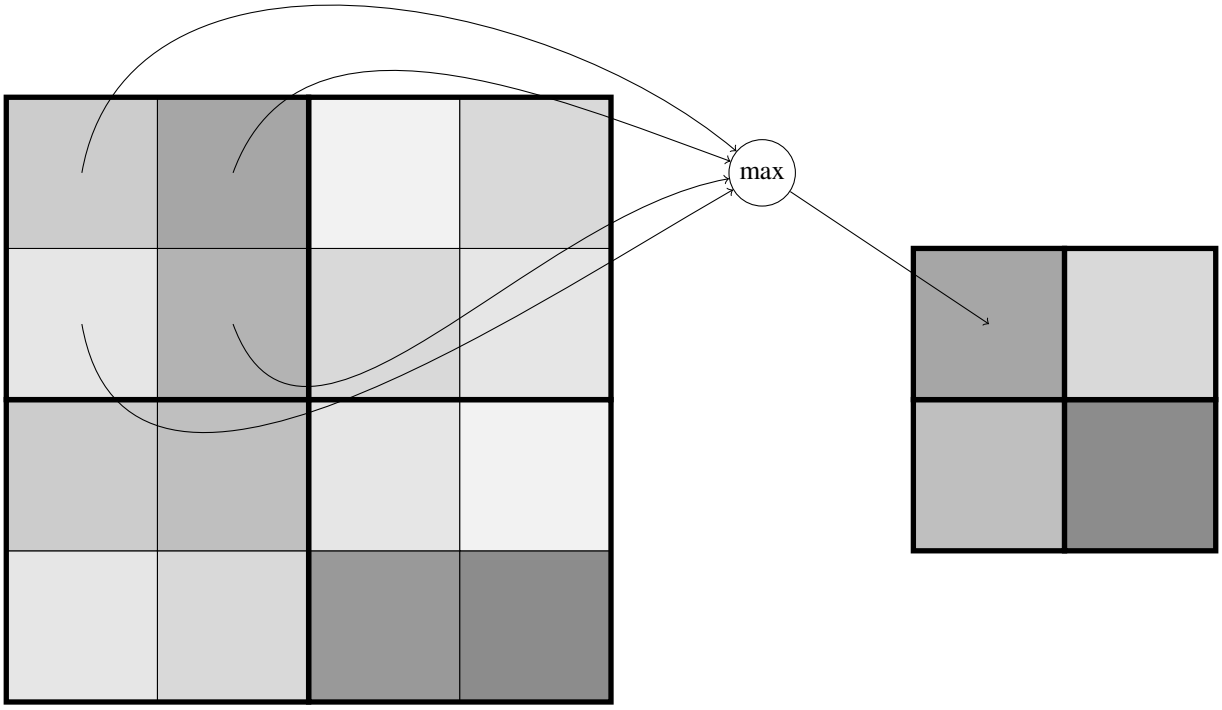
Figure6.3: Visualisation de l'effet du *padding* (source: V. Dumoulin, F. Visin - A guide to convolution arithmetic for deep learning). Gauche: sans *padding*, droite: avec *padding*.

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

model = Sequential([
    InputLayer(input_shape=(32, 32, 1)),
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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()
```

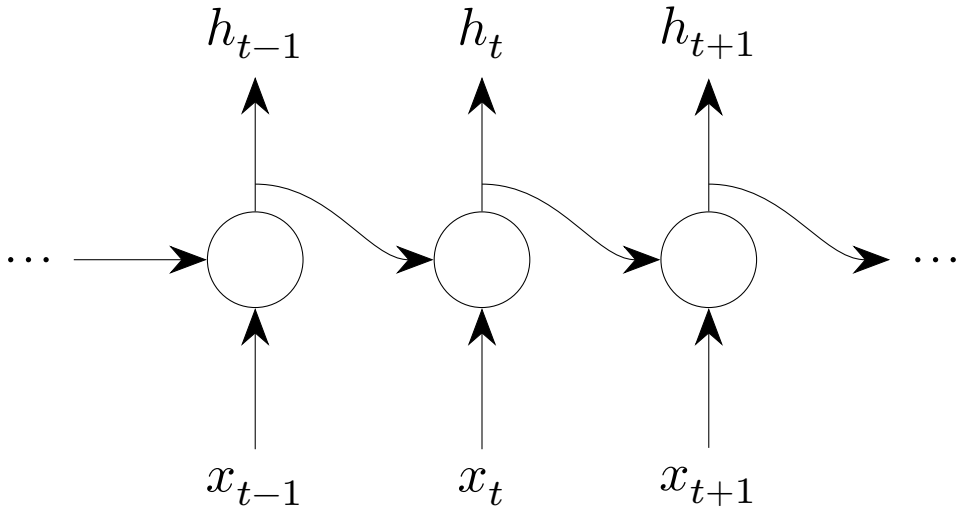
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         156

 max_pooling2d (MaxPooling2D  (None, 14, 14, 6)        0
 )

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

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 16)         0
 2D)

 flatten (Flatten)           (None, 400)               0

 dense (Dense)               (None, 120)               48120

 dense_1 (Dense)             (None, 84)                10164

 dense_2 (Dense)             (None, 10)                850

=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
_________________________________________________________________
```

6.3 Références

CHAPTER 7

RECURRENT NEURAL NETWORKS

Recurrent neural networks (RNNs) proceed by processing elements of a time series one at a time. Typically, at time t, a recurrent block will take both the current input x_t and a hidden state h_{t-1} that aims at summarizing the key information from past inputs $\{x_0, ..., x_{t-1}\}$, and will output an updated hidden state h_t:



There exist various recurrent modules that mostly differ in the way h_t is computed.

7.1 « Vanilla » RNNs

The basic formulation for a RNN block is as follows:

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

where W_h is a weight matrix associated to the processing of the previous hidden state, W_x is another weight matrix associated to the processing of the current input and b is a bias term.

Note here that W_h, W_x and b are not indexed by t, which means that they are **shared across all timestamps**.

An important limitation of this formula is that it easily fails at capturing long-term dependencies. To better understand why, one should remind that the parameters of these networks are optimized through stochastic gradient descent algorithms.

To simplify notations, let us consider a simplified case in which h_t and x_t are both scalar values, and let us have a look at what the actual gradient of the output h_t is, with respect to W_h (which is then also a scalar):

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

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

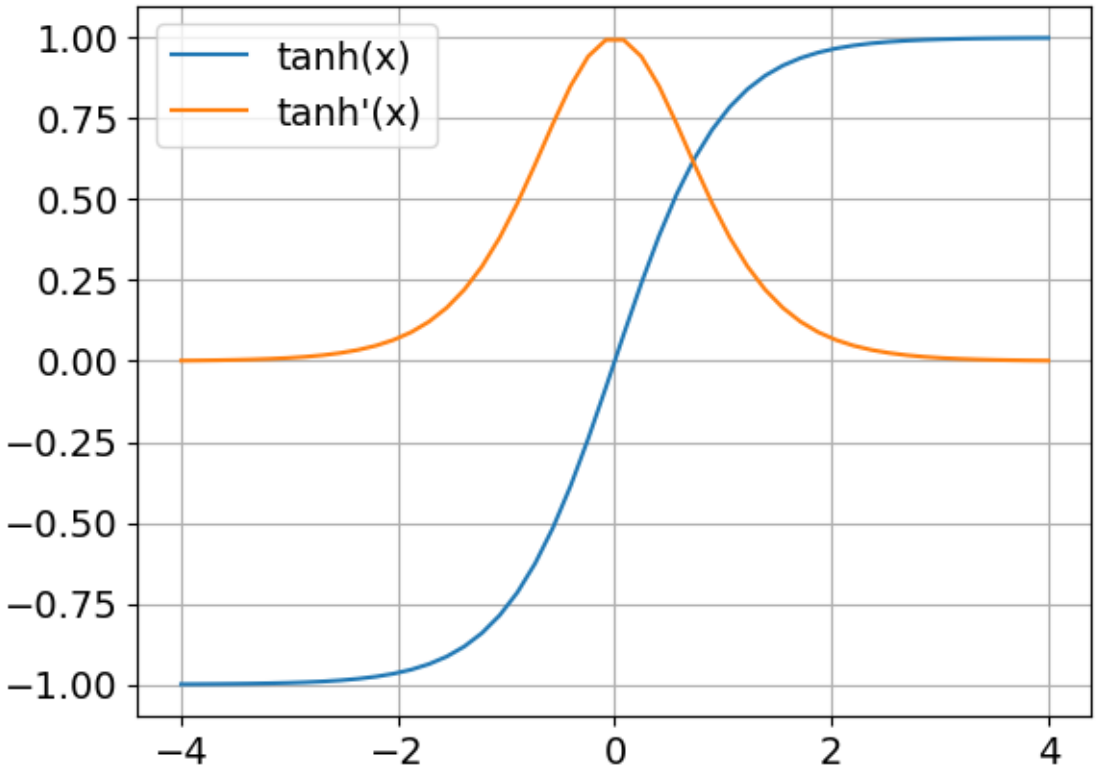
$$\frac{\partial o_t}{\partial W_h} = h_{t-1} + W_h \cdot \frac{\partial h_{t-1}}{\partial W_h} \,. \tag{7.3}$$

Here, the form of $\frac{\partial h_{t-1}}{\partial W_h}$ will be similar to that of $\nabla_{W_h}(h_t)$ above, and, in the end, one gets:

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

In other words, the influence of $h_{t'}$ will be mitigated by a factor $W_h^{t-t'-1}\tanh'(o_{t'+1}) \cdot \cdots \cdot \tanh'(o_t)$.

Now recall what the tanh function and its derivative look like:



One can see how quickly gradients gets close to 0 for inputs larger (in absolute value) than 2, and having multiple such terms in a computation chain will likely make the corresponding terms vanish.

In other words, the gradient of the hidden state at time t will only be influenced by a few of its predecessors $\{h_{t-1}, h_{t-2}, \dots\}$ and long-term dependencies will be ignored when updating model parameters through gradient descent. This is an occurrence of a more general phenomenon known as the **vanishing gradient** effect.

7.2 Long Short-Term Memory

The Long Short-Term Memory (LSTM, [Hochreiter and Schmidhuber, 1997]) blocks have been designed as an alternative recurrent block that aims at mitigating this vanishing gradient effect through the use of gates that explicitly encode pieces of information that should (resp. should not) be kept in computations.

Gates in neural networks

In the neural networks terminology, a gate $g \in [0,1]^d$ is a vector that is used to filter out information from an incoming feature vector $v \in \mathbb{R}^d$ such that the result of applying the gate is: $g \odot v$ where $\odot$ is the element-wise product. The gate g will hence tend to remove part of the features in v (those corresponding to very low values in g).

In these blocks, an extra state is used, referred to as the cell state C_t. This state is computed as:

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

where f_t is the forget gate (which pushes the network to forget about useless parts of the past cell state), i_t is the input gate and $\tilde{C}_t$ is an updated version of the cell state (which, in turn, can be partly censored by the input gate).

Let us delay for now the details about how these 3 terms are computed, and rather focus on how the formula above is significantly different from the update rule of the hidden state in vanilla RNNs. Indeed, in this case, if the network learns

so (through f_t), the full information from the previous cell state C_{t-1} can be recovered, which would allow gradients to flow through time (and not vanish anymore).

Then, the link between the cell and hidden states is:

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

In words, the hidden state is the tanh-transformed version of the cell state, further censored by an output gate o_t.

All gates used in the formulas above are defined similarly:

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

where σ is the sigmoid activation function (which has values in $[0, 1]$) and $[h_{t-1}, x_t]$ is the concatenation of h_{t-1} and x_t features.

Finally, the updated cell state $\tilde{C}_t$ is computed as:

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

Many variants over these LSTM blocks exist in the literature that still rely on the same basic principles.

7.3 Gated Recurrent Unit

A slightly different parametrization of a recurrent block is used in the so-called Gatted Recurrent Unit (GRU, [Cho *et al.*, 2014]).

GRUs also rely on the use of gates to (adaptively) let information flow through time. A first significant difference between GRUs and LSTMs, though, is that GRUs do not resort to the use of a cell state. Instead, the update rule for the hidden state is:

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

where z_t is a gate that balances (per feature) the amount of information that is kept from the previous hidden state with the amount of information that should be updated using the new candidate hidden state $\tilde{h}_t$, computed as:

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

where r_t is an extra gate that can hide part of the previous hidden state.

Formulas for gates z_t and r_t are similar to those provided for f_t, i_t and o_t in the case of LSTMs.

A graphical study of the ability of these variants of recurrent networks to learn long-term dependencies is provided in [Madsen, 2019].

7.4 Conclusion

In this chapter, we have reviewed neural network architectures that are used to learn from time series datasets. Because of time constraints, we have not tackled attention-based models in this course. We have presented convolutional models that aim at extracting discriminative local shapes in the series and recurrent models that rather leverage the notion of sequence. Concerning the latter, variants that aim at facing the vanishing gradient effect have been introduced. Note that recurrent models are known to require more training data than their convolutional counterparts in order to learn meaningful representations.

BIBLIOGRAPHY

[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.