Reconnaissance d'images avec des réseaux denses

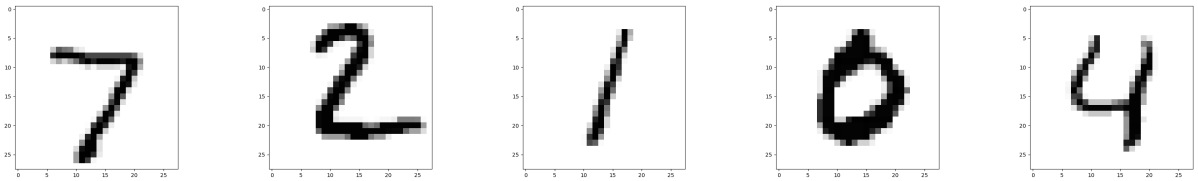
Objectifs [a] *:*

- *Codage d'une image simple en niveaux de gris et de ses labels*
- *Introduction de la fonction d'activation "softmax" et de la fonction d'erreur "entropie croisée catégorielle"*
- *Introduction de l'optimisation "descente de gradient stochastique"*
- *Application, par un réseau dense, à la reconnaissance d'image - comprendre les limitations de la méthode*

a. Version 2023 adapté BUT 3 Informatique (Orléans), inspiré du livre d'Arnaud Bodin et François Recher

1. Données et composition du réseau

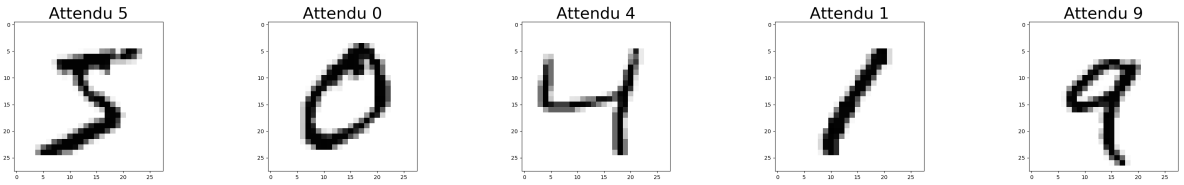
Il s'agit de reconnaître de façon automatique des chiffres écrits à la main.



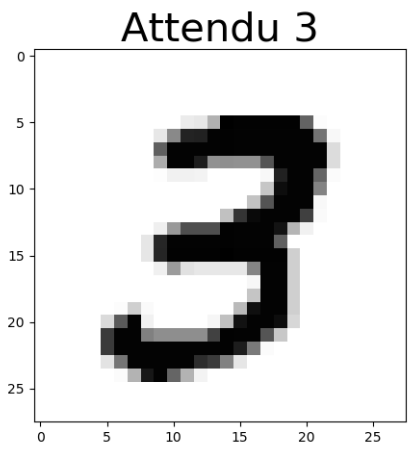
C'est l'un des succès historiques des réseaux de neurones qui permet par exemple le tri automatique du courrier par lecture du code postal.

1.1. Base MNIST

Un élément essentiel de l'apprentissage automatique est de disposer de données d'apprentissage nombreuses et de qualité. Une telle base est la base MNIST dont voici les premières images.



Une donnée est constituée d'une image et du chiffre attendu.



Plus en détails :

- La base est formée de 60 000 données d'apprentissage et de 10 000 données de test.
- Chaque donnée est de la forme : [une image, le chiffre attendu].
- Chaque image est de taille 28×28 pixels, chaque pixel contenant un des 256 niveaux de gris (numérotés de 0 à 255).

Ces données sont accessibles très simplement avec *tensorflow*/*keras* :

```
from tensorflow.keras.datasets import mnist
(X_train_data,Y_train_data),(X_test_data,Y_test_data) = mnist.load_data()
```

Il faut passer un peu de temps à comprendre et à manipuler les données. `X_train_data[i]` correspond à une image et `Y_train_data[i]` au chiffre attendu pour cette image. Nous parlerons plus tard des données de test. On renvoie au fichier `tf2_chiffres_data.py` pour une exploration de la base. Noter que l'on peut afficher une image à l'aide de *matplotlib* par la commande :

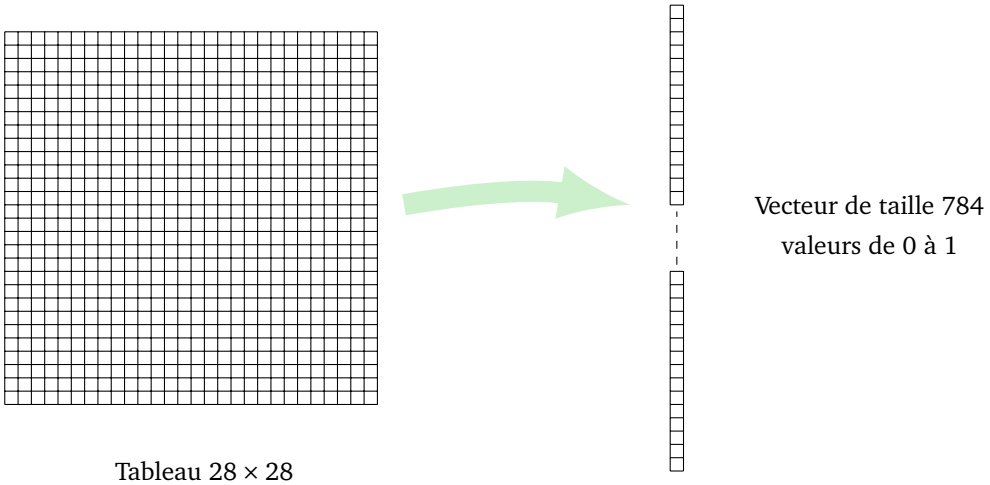
```
plt.imshow(X_train_data[i], cmap='Greys')
```

suivie de `plt.show()`.

1.2. Traitement des données

Pour une utilisation par un réseau de neurones nous devons d'abord transformer les données.

Donnée d'entrée. En entrée du réseau de neurones, nous devons avoir un vecteur. Au départ chaque image est un tableau de taille 28×28 ayant des entrées entre 0 et 255. Nous la transformons en un vecteur de taille $784 = 28^2$ et nous normalisons les données dans l'intervalle $[0, 1]$ (en divisant par 255).



Ainsi, une entrée X est un « vecteur-image », c'est-à-dire un vecteur de taille 784 représentant une image.

Donnée de sortie. Notre réseau de neurones ne va pas renvoyer le chiffre attendu, mais une liste de 10 probabilités. Ainsi chaque chiffre doit être codé par une liste de 0 et de 1.

- 0 est codé par $(1,0,0,0,0,0,0,0,0,0)$,
- 1 est codé par $(0,1,0,0,0,0,0,0,0,0)$,
- 2 est codé par $(0,0,1,0,0,0,0,0,0,0)$,
- ...
- 9 est codé par $(0,0,0,0,0,0,0,0,0,1)$.

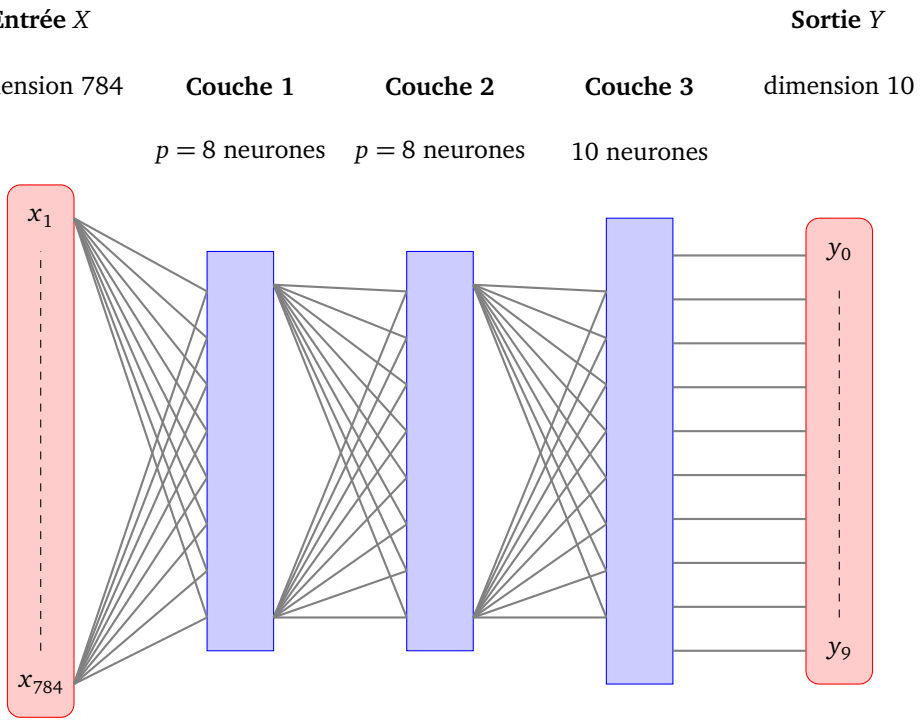
Fonction. Nous cherchons une fonction $F : \mathbb{R}^{784} \to \mathbb{R}^{10}$, qui à un vecteur-image associe une liste de probabilités, telle que $F(X_i) \simeq Y_i$ pour nos données transformées (X_i, Y_i), $i = 1, \ldots, 60\,000$.
Par exemple la fonction F, évaluée sur un vecteur-image X peut renvoyer

$$F(X) = (0.01, 0.04, 0.03, 0.01, 0.02, 0.22, 0.61, 0.02, 0.01, 0.01, 0.02).$$

Dans ce cas, le nombre le plus élevé est 0.61 au rang 6, cela signifie que notre fonction F prédit le chiffre 6 avec une probabilité de 61%, mais cela pourrait aussi être le chiffre 5 qui est prédit à 22%. Les autres chiffres sont peu probables.

1.3. Réseau

Nous allons construire un réseau de neurones qui produira une fonction $F : \mathbb{R}^{784} \to \mathbb{R}^{10}$. L'architecture est composée de 3 couches. En entrée nous avons un vecteur de taille 784. La première et la seconde couches sont composées chacune de $p = 8$ neurones. La couche de sortie est formée de 10 neurones, un pour chacun des chiffres. Pour les fonctions d'activation nous utiliserons, pour les deux premières couches, la fonction sigmoid (σ) et pour la dernière la fonction sofmax (utilisée dans un but de classification) que nous définissons dans le paragraphe suivant. Aussi nous introduirons la fonction d'erreur la plus utilisée pour la reconnaissance d'image : l'entropie croisée catégorielle.

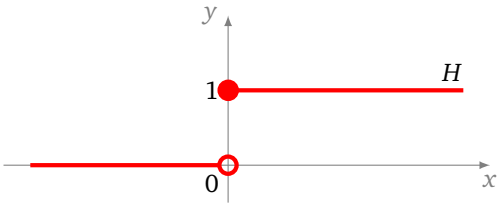


2. Nouveauté : fonctions d'activation, d'erreurs et optimisation

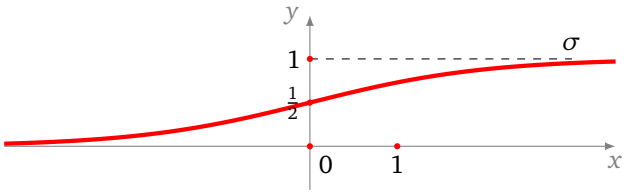
2.1. Les fonctions d'activation : sigmoid et softmax

Rappel : fonctions H/σ

On souhaite pouvoir distinguer une image d'un chat (valeur 0) de celle d'un chien (valeur 1). La première idée serait d'utiliser la fonction marche de Heaviside qui répondrait très clairement à la question, mais nous allons voir que la fonction σ est plus appropriée.



$$\begin{cases} H(x) = 0 & \text{si } x < 0 \\ H(x) = 1 & \text{si } x \geqslant 0 \end{cases}$$



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

On rappelle d'abord les qualités de la fonction σ par rapport à la fonction marche de Heaviside :

- La fonction σ est dérivable en tout point avec une dérivée non identiquement nulle, ce qui est important pour la descente de gradient. A contrario la fonction H est de dérivée partout nulle (sauf en 0 où elle n'est pas dérivable).
- La fonction σ prend toutes les valeurs entre 0 et 1, ce qui permet de nuancer le résultat. Par exemple si la valeur est 0.92, on peut affirmer que l'image est plutôt celle d'un chien.

On interprète donc une valeur p entre 0 et 1 renvoyée par σ comme une probabilité, autrement dit un degré de certitude. Évidemment, si l'on souhaite à chaque fois se positionner, on peut répondre : 0 si $\sigma(x) < \frac{1}{2}$ et 1 sinon. Cela revient à composer σ par la fonction $H(x - \frac{1}{2})$.

Fonction d'activation softmax

Comment construire une fonction qui permette de décider non plus entre deux choix possibles, mais parmi un nombre fixé, comme par exemple lorsqu'il faut classer des images en 10 catégories ? Une solution est d'obtenir k valeurs $(x_1, \ldots, x_k)$. On rattache $(x_1, \ldots, x_k)$ à l'une des k catégories données par un vecteur de longueur k :

- $(1, 0, 0, \ldots, 0)$ première catégorie,
- $(0, 1, 0, \ldots, 0)$ deuxième catégorie,
- $\ldots$
- $(0, 0, 0, \ldots, 1)$ k-ème et dernière catégorie.

Argmax. La fonction argmax renvoie le rang pour lequel le maximum est atteint. Par exemple si $X = (x_1, x_2, x_3, x_4, x_5) = (3, 1, 8, 6, 0)$ alors argmax(X) vaut 3, car le maximum $x_3 = 8$ est atteint au rang 3. Cela correspond donc à la catégorie $(0, 0, 1, 0, 0)$.

Softmax. Pour $X = (x_1, x_2, \ldots, x_k)$ on note

$$\sigma_i(X) = \frac{e^{x_i}}{e^{x_1} + e^{x_2} + \cdots + e^{x_k}}.$$

Proposition 1.
Pour chaque $i = 1, \ldots, k$ on a $0 < \sigma_i(X) \leqslant 1$ et

$$\sigma_1(X) + \sigma_2(X) + \cdots + \sigma_k(X) = 1.$$

Comme la somme des $\sigma_i(X)$ vaut 1, on interprète la valeur $\sigma_i(X)$ comme une probabilité p_i. L'ensemble des valeurs $(\sigma_i(X), i \leqslant k)$ s'appelle la **distribution (ou loi) de** X. Par exemple avec $X = (3,1,8,6,0)$ alors

$$p_1 = \sigma_1(X) \simeq 0.0059 \quad p_2 = \sigma_2(X) \simeq 0.0008 \quad p_3 \simeq 0.8746 \quad p_4 \simeq 0.1184 \quad p_5 \simeq 0.0003$$

On note que la valeur maximale est obtenue au rang 3. On obtient aussi ce rang 3 en composant par la fonction argmax.

2.2. Loss function (fonction d'erreur) : Categorical crossentropy

La fonction de perte que nous avons vu jusqu'à maintenant était l'erreur de carré moyenne, celle-ci mesure bien des distances au sens classique du terme. Pour la reconnaissance d'image nous voulons mesurer des "distances" un peu différentes, associées à des probabilités. En effet nous souhaitons mesurer la différence entre la distribution de probabilité prédite par un modèle de réseau de neurones et la distribution de probabilité réelle des étiquettes (ou label) de catégorie (ou classe). Les "étiquettes de catégories" indiquent à quelle classe chaque exemple de données appartient, comme "3" pour une image de chiffre "3" ou "7" pour une image de chiffre "7".

Ainsi la "categorical crossentropy" (entropie croisée catégorielle) est une fonction de perte couramment utilisée dans l'apprentissage automatique, en particulier dans les tâches de classification multiclasse. Plus précisément, elle quantifie l'erreur en calculant la somme des logarithmes négatifs des probabilités prédites pour les classes réelles :

$$\text{Categorical Crossentropy} = -\sum_{i \leqslant k} p_i \ln(q_i) \tag{1}$$

où p_i représente la probabilité réelle de la classe i (typiquement p_i vaut 0 zero ou 1), et q_i la probabilité produite pour la classe i par le réseau (si on pense à l'exemple du paragraphe précédent $q_i = \sigma(X_i)$) enfin k est le nombre de catégories. Cette fonction de perte favorise la convergence du modèle vers des prédictions de probabilité proches de zéro pour les classes incorrectes et proches de un pour la classe correcte.

> **Exemple.**
> : Supposons que nous ayons un modèle de classification de chiffres manuscrits (0 à 9) et que nous souhaitons prédire le chiffre "3". Si la probabilité réelle d'être un "3" est de 1 (car il s'agit d'un "3" réel) et que le modèle prédit une probabilité de 0,9 pour "3", alors la perte "categorical crossentropy" serait de $-\ln(0.9) \simeq 0.046$, soit une petite valeur. Cependant, si le modèle prédit une probabilité de 0,1 pour "3", la perte serait beaucoup plus élevée $-\ln(0.1) \simeq -2.3$, ce qui reflète une prédiction incorrecte.
> Cette fonction de perte est essentielle pour entraîner des réseaux de neurones dans des tâches de classification multiclasse/multicatégorielle en minimisant l'erreur de prédiction.

Exercice 1.
Calculer l'entropie croisée catégorielle, pour $p_1 = p_2 = p_3 = 0$ et $p_4 = 1$ et
1) $q_1 = q_2 = q_3 = q_4 = 0.25$,
2)$q_1 = q_2 = 0.01$, $q_3 = q_4 = 0.49$,
3) $q_1 = 0.9$, $q_2 = q_3 = 0.03$, $q_4 = 0.04$.
Que remarquez vous ?

2.3. Optimisation usuelle : Descente de gradiant stochastique (sgd)

La descente de gradient stochastique (abrégée en *sgd*) est une façon d'optimiser les calculs de la descente de gradient pour une fonction d'erreur associée à une grande série de données. Au lieu de calculer un

gradient (compliqué) et un nouveau point pour l'ensemble des données, on calcule un gradient (simple) et un nouveau point par donnée, il faut répéter ce processus pour chaque donnée.

Petits pas à petits pas

Revenons à l'objectif visé par la régression linéaire.
On considère des données (X_i, y_i), $i = 1, \dots, N$ où $X_i \in \mathbb{R}^\ell$ et $y_i \in \mathbb{R}$. Ces données proviennent d'observations ou d'expérimentations.
Il s'agit de trouver une fonction $F : \mathbb{R}^\ell \to \mathbb{R}$ qui modélise au mieux ces données, c'est-à-dire telle que

$$F(X_i) \simeq y_i.$$

Pour ***l'entrée*** X_i, la valeur y_i est la ***sortie attendue***, alors que $F(X_i)$ est la ***sortie produite*** par notre modèle.
Pour mesurer la pertinence de la fonction F, on introduit la fonction d'***erreur totale*** qui mesure l'écart entre la sortie attendue et la sortie produite :

$$E = \sum_{i=1}^{N} E_i = \sum_{i=1}^{N} (y_i - F(X_i))^2.$$

Cette erreur totale est une somme d'***erreurs locales*** :

$$E_i = (y_i - F(X_i))^2.$$

Le but du problème est de déterminer la fonction F qui minimise l'erreur E. Par exemple, dans le cas de la régression linéaire, il fallait trouver les paramètres a et b pour définir $F(x) = ax + b$, ou bien, pour deux variables, les paramètres a, b, c pour définir $F(x, y) = ax + by + c$.
Considérons une fonction erreur $E : \mathbb{R}^n \to \mathbb{R}$ qui dépend de n paramètres $a_1, \dots, a_n$ (qui définissent l'expression de la fonction F).

Descente de gradient classique. Pour minimiser l'erreur et déterminer les meilleurs paramètres, on peut appliquer la méthode du gradient classique.
On part d'un point $P_0 = (a_1, \dots, a_n) \in \mathbb{R}^n$, puis on applique la formule de récurrence :

$$P_{k+1} = P_k - \delta \operatorname{grad} E(P_k).$$

Pour appliquer cette formule, il faut calculer des gradients $\operatorname{grad} E(P_k)$, or

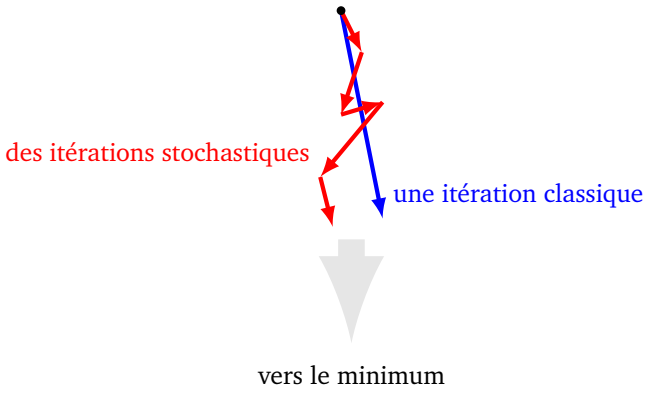
$$\operatorname{grad} E(P_k) = \sum_{i=1}^{N} \operatorname{grad} E_i(P_k).$$

Il faut donc calculer une somme de N termes à chaque itération, ce qui pose des problèmes d'efficacité pour de grandes valeurs de N.

Descente de gradient stochastique.
Pour diminuer la quantité de calculs, l'idée est de considérer à chaque itération un seul gradient E_i à la place de E. C'est-à-dire :

$$P_{k+1} = P_k - \delta \operatorname{grad} E_i(P_k)$$

pour *une seule erreur* E_i (correspondant à la donnée numéro i). L'itération suivante se basera sur l'erreur E_{i+1}.
Quel est l'intérêt de cette méthode ? Dans la méthode de gradient classique, on calcule à chaque itération un « gros » gradient (associé à la totalité des N données) qui nous rapproche d'un grand pas vers le minimum. Ici on calcule N « petits » gradients qui nous rapprochent du minimum.



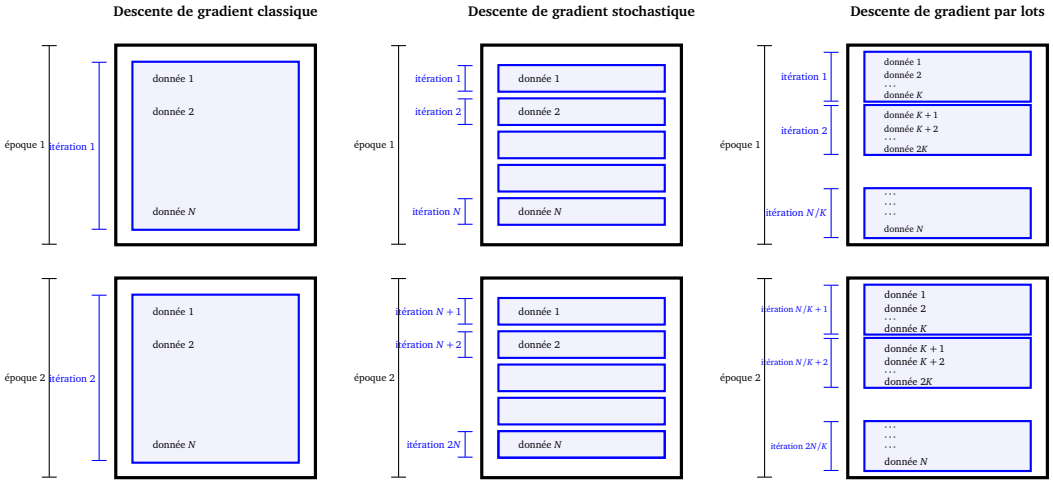
Voici les premières itérations de cet algorithme.

- On part d'un point P_0.
- On calcule $P_1 = P_0 - \delta \operatorname{grad} E_1(P_0)$. C'est la formule du gradient, mais seulement pour l'erreur locale E_1 (juste à partir de la première donnée (X_1, y_1)).
- On calcule $P_2 = P_1 - \delta \operatorname{grad} E_2(P_1)$. C'est la formule du gradient, mais seulement pour l'erreur locale E_2.
- On itère encore et encore.
- On calcule $P_N = P_{N-1} - \delta \operatorname{grad} E_N(P_{N-1})$. C'est la formule du gradient, mais seulement pour l'erreur locale E_N. À ce stade de l'algorithme, nous avons tenu compte de toutes les données.
- On calcule $P_{N+1} = P_N - \delta \operatorname{grad} E_1(P_N)$. On recommence pour P_N et l'erreur locale E_1.
- Etc. On s'arrête au bout d'un nombre d'étapes fixé à l'avance ou lorsque l'on est suffisamment proche du minimum.

Exercice 2.
On considère l'erreur des moindres carrés, prenons $N = 3$. Donner les expressions de :
1) $P_{k+1} = P_k - \delta \operatorname{grad} E(P_k)$ (descente de gradient classique)
2) $P_{k+1} = P_k - \delta \operatorname{grad} E_i(P_k)$ (descente de gradient stochastique, étape i).

Descente par lots

Il existe une méthode intermédiaire entre la descente de gradient classique (qui tient compte de toutes les données à chaque itération) et la descente de gradient stochastique (qui n'utilise qu'une seule donnée à chaque itération).
La descente de gradient par ***lots*** (ou ***mini-lots***, *mini-batch*) est une méthode intermédiaire : on divise les données par paquets de taille K. Pour chaque paquet (appelé « lot »), on calcule un gradient et on effectue une itération.
Au bout de N/K itérations, on a parcouru tout le jeu de données : cela s'appelle une ***époque***.



La formule est donc

$$P_{k+1} = P_k - \delta \operatorname{grad}(E_{j_0+1} + E_{j_0+2} + \cdots + E_{j_0+K})(P_k).$$

Pour P_{k+2}, on repart de P_{k+1} et on utilise le gradient de la fonction $E_{j_0+K+1} + E_{j_0+K+2} + \cdots + E_{j_0+2K}$.

Remarque.

- Pour $K = 1$, c'est exactement la descente de gradient stochastique. Pour $K = N$, c'est la descente de gradient classique.
- Cette méthode combine le meilleur des deux mondes : la taille des données utilisées à chaque itération peut être adaptée à la mémoire et le fait de travailler par lots évite les pas erratiques de la descente stochastique pure.
- On peut par exemple choisir $2 \leqslant K \leqslant 32$ et profiter du calcul parallèle en calculant $\operatorname{grad}(E_1 + \cdots + E_K)$, par le calcul de chacun des $\operatorname{grad} E_i$ sur K processeurs, puis en additionnant les résultats.
- Il est d'usage de mélanger au hasard les données (X_i, y_i) avant chaque époque.

Exercice 3.

On considère l'erreur des moindres carrés. Donner l'expression :

1) $P_{k+1} = P_k - \delta \operatorname{grad} E_1(P_k) - \delta \operatorname{grad} E_2(P_k)$ (Batch 1)

2) $P_{k+2} = P_{k+1} - \delta \operatorname{grad} E_3(P_{k+1}) - \delta \operatorname{grad} E_4(P_{k+1})$ (Batch 2).

Pour simplifier on pourra prendre $N = 4$. C'est une descente de gradient stochastique par mini-batch de taille $K = 2$.

3. Code du réseau - entrainement et test

3.1. Le code

Voici le code complet du programme qui sera commenté plus loin.

```
import numpy as np
from tensorflow import keras
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

### Partie A - Les données

from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Téléchargement des données
(X_train_data,Y_train_data),(X_test_data,Y_test_data) = mnist.load_data()

N = X_train_data.shape[0]  # N = 60 000 données

# Données d'apprentissage X
X_train = np.reshape(X_train_data,(N,784))  # vecteur image
X_train = X_train/255  # normalisation

# Données d'apprentissage Y vers une liste de taille 10
Y_train = to_categorical(Y_train_data, num_classes=10)
```

```
# Données de test
X_test = np.reshape(X_test_data,(X_test_data.shape[0],784))
X_test = X_test/255
Y_test = to_categorical(Y_test_data, num_classes=10)

### Partie B - Le réseau de neurones

p = 8
modele = Sequential()

# Première couche : p neurones (entrée de dimension 784 = 28x28)
modele.add(Dense(p, input_dim=784, activation='sigmoid'))

# Deuxième couche : p neurones
modele.add(Dense(p, activation='sigmoid'))

# Couche de sortie : 10 neurones (un par chiffre)
modele.add(Dense(10, activation='softmax'))

# La fonction d'activation 'softmax' sera décrite dans le paragraphe suivant.

# Choix de la méthode de descente de gradient
modele.compile(loss='categorical_crossentropy',
               optimizer='sgd',
               metrics=['accuracy'])

# La fonction d'erreur 'categorical_crossentropy' est décrite dans le paragraphe suivant.
# L'optimisation 'sgd' est décrite dans le paragraphe suivant.
# 'accuracy' est décrite dans le paragraphe suivant.

print(modele.summary())
```

3.2. Commentaires sur le code

Partie A - Les données

Les données d'apprentissage sont téléchargées facilement par une seule instruction. Elles regroupent $N = 60\,000$ données d'apprentissage (*train*) et 10 000 données de test. Les données sont de la forme (X_i, Y_i) où X_i est une image (un tableau 28×28 d'entiers de 0 à 255) et Y_i est le chiffre correspondant (de 0 à 9). Les données X_i sont transformées en un vecteur de taille 784 (fonction `reshape()`) et ses coefficients sont ramenés dans l'intervalle $[0, 1]$ par division par 255. Ainsi `X_train` est maintenant une liste *numpy* de 60 000 vecteurs de taille 784.
Chaque donnée Y_i est transformée en une liste de longueur 10 du type `[0,0,...,0,1,0,...,0]` avec le 1 à la place du chiffre attendu. On utilise ici la fonction `to_categorical()`.

Partie B - Le réseau de neurones

Notre réseau est composé de 3 couches. La première couche contient $p = 8$ neurones et reçoit en entrée 784 valeurs (une pour chaque pixel de l'image). La seconde couche contient aussi $p = 8$ neurones. La troisième

couche contient 10 neurones (le premier pour détecter le chiffre 0, le deuxième pour le chiffre 1,...). Pour les deux premières couches la fonction d'activation est la fonction σ. Pour la couche de sortie, la fonction d'activation est la fonction *softmax* qui est adaptée au problème. Ce qui fait que la couche de sortie renvoie une liste de 10 nombres dont la somme est 1 et qui correspond à une liste de probabilités.
La fonction d'erreur (*loss*) adaptée au problème s'appelle `categorical_crossentropy`. La méthode de minimisation de l'erreur choisie est la descente de gradient stochastique.
La valeur `accuracy` du paramètre `metrics` indique que l'on souhaite en plus mesurer la précision des résultats (cela ne change rien pour la descente de gradient qui dépend uniquement de la fonction d'erreur et pas de cette précision).

3.3. Dernières remarques

On lance le calcul des poids par la fonction `fit()` :

```
modele.fit(X_train, Y_train, batch_size=32, epochs=100)
```

- Les calculs sont effectués selon la méthode de descente de gradient choisie auparavant.
- Les données d'apprentissage utilisées sont `X_train` (valeurs de départ) et `Y_train` (valeurs d'arrivée attendues).
- L'option `batch_size` précise la taille de l'échantillon :
  — une descente de gradient stochastique pure : `batch_size=1`,
  — une descente de gradient sur la totalité des N données : `batch_size=N`,
  — ou toute valeur intermédiaire : par défaut `batch_size=32`.
- L'option `epochs` détermine le nombre d'étapes dans la méthode de gradient. Si K est la taille d'un lot (valeur de `batch_size`) et N la taille des données alors le nombre d'étapes par époque est N/K.
- Il existe une option `verbose` qui permet d'afficher plus ou moins de détails à chaque époque (0 : rien, 1 : barre de progression, 2 : numéro de l'époque).

On peut vouloir mesurer d'autres choses que la fonction d'erreur (moindre carré, entropie catégorielle). Par exemple l'option de la fonction `compile()` :

```
metrics=['accuracy']
```

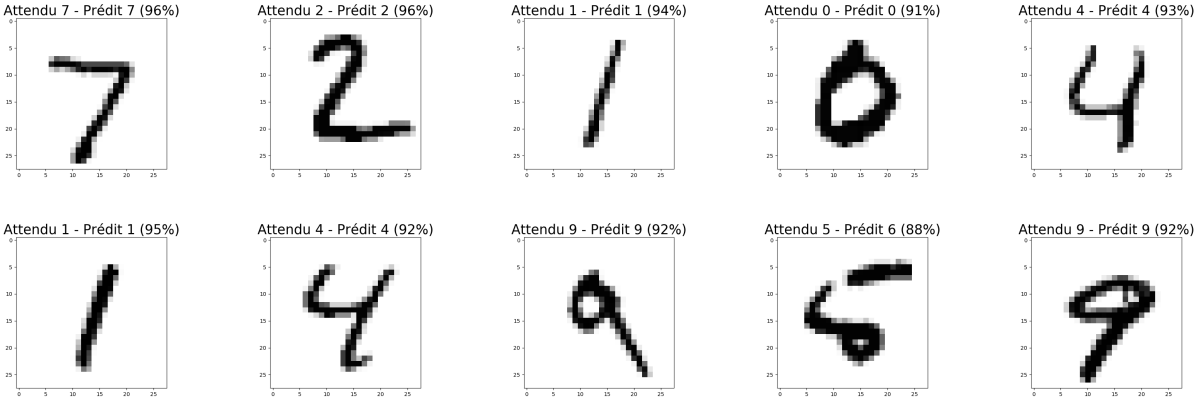
va en plus mémoriser la précision du modèle (sous la forme d'un pourcentage de réussite). Par exemple, si on doit classer des images en deux catégories (chat/0 et chien/1) alors la fonction d'erreur est un outil indispensable pour notre problème, mais le résultat est mesuré concrètement par le pourcentage d'images correctement identifiées.
Pour la calculer on compare les prédictions du modèle aux données réelles (étiquettes) dans l'ensemble de données de test. Le nombre de bonnes prédictions (les prédictions correctes) est représenté par M dans la formule de l'accuracy, et N est le nombre total d'exemples dans l'ensemble de données de test. On a alors : `accuracy`$= M/N$.
Note : pourquoi ne pas prendre la précision comme fonction d'erreur puisque c'est ce qui nous intéresse ? Tout simplement parce que ce n'est pas un fonction différentiable, il n'y a donc pas de méthode de gradient pour la minimiser.

4. Résultats

On lance maintenant la procédure l'apprentissage, puis on analyse les résultats.

```
### Partie C - Calcul des poids par descente de gradient

modele.fit(X_train, Y_train, batch_size=32, epochs=40)

### Partie D - Résultats

resultat = modele.evaluate(X_test, Y_test, verbose=0)
print('Valeur de l''erreur sur les données de test (loss):', resultat[0])
print('Précision sur les données de test (accuracy):', resultat[1])
```

Remarque : `epochs=40` signifie que l'on se sert 40 fois des mêmes données.
On estime la performance de notre réseau sur les données de test. Voici les premiers résultats.



Le réseau prédit correctement 9 valeurs sur 10. En fait, la fonction associée au réseau renvoie une liste de probabilités. Le chiffre prédit est celui qui a la plus forte probabilité. Par exemple pour la première image (en haut à gauche) la valeur renvoyée est :

$$Y_0 = (0.001, 0.000, 0.000, 0.008, 0.002, 0.005, 0.000, 0.965, 0.000, 0.020).$$

On en déduit la prédiction du chiffre 7 avec une forte probabilité de 96%.
L'avant-dernière image conduit à une mauvaise prédiction : la fonction prédit le chiffre 6 alors que le résultat attendu est le chiffre 5.
Les calculs de la descente de gradient sont faits avec une fonction d'erreur qu'il s'agit de minimiser. Par contre cette fonction d'erreur n'est pas pertinente pour évaluer la qualité de la modélisation. On préfère ici calculer la ***précision*** (*accuracy*) qui correspond à la proportion de chiffres détectés correctement.

Exercice 4.
Calculer l'entropie croisée catégorielle pour l'exemple ci-dessus.

Voici quelques résultats pour différentes tailles du réseau (couche 1 avec p neurones, couche 2 avec aussi p neurones, couche 3 avec 10 neurones) :

p	neurones	poids	précision
8	26	6442	90.0%
10	30	8070	91.4%
20	50	16 330	93.4%
50	110	42 310	94.4%

Sans trop d'efforts on obtient donc une précision de 95%. C'est-à-dire que 95 fois sur 100 le chiffre prédit est le chiffre correct.
Il est important de mesurer la précision sur les données de test qui sont des données qui n'ont pas été utilisées lors de l'apprentissage. Le réseau n'a donc pas appris par cœur les données d'apprentissage, mais a réussi à dégager un schéma, validé sur des données indépendantes.

4.1. Détails sur le code

Reprenons pas à pas le programme et donnons quelques explications.

Partie C - Calcul des poids par descente de gradient

La fonction `fit()` lance la descente de gradient (avec une initialisation aléatoire des poids). L'option `batch_size` détermine la taille du lot (*batch*). On indique aussi le nombre d'époques à effectuer (avec `epochs=40`, chacune des $N = 60\,000$ données sera utilisée 40 fois, mais comme chaque étape regroupe un lot de 32 données, il y $40 \times N/32$ étapes de descente de gradient).

Partie D - Résultats

On insiste sur le fait que la performance du réseau avec les poids calculés doit être mesurée sur les données de test et non sur les données d'apprentissage.
La fonction `evaluate()` renvoie la valeur de la fonction d'erreur (qui est la fonction minimisée par la descente de gradient mais qui n'a pas de signification tangible). Ici la fonction renvoie aussi la précision (car on l'avait demandée en option dans la fonction `compile()`).

Un peu plus de résultats

Pour l'instant, nous avons juste mesuré l'efficacité globale du réseau. Il est intéressant de vérifier à la main les résultats. Les instructions ci-dessous calculent les prédictions pour toutes les données de test (première ligne). Ensuite, pour une donnée particulière, on compare le chiffre prédit et le chiffre attendu.

```
# Prédiction sur les données de test
Y_predict = modele.predict(X_test)

# Un exemple
i = 8  # numéro de l'image

chiffre_predit = np.argmax(Y_predict[i]) # prédiction par le réseau

print("Sortie réseau", Y_predict[i])
print("Chiffre attendu :", Y_test_data[i])
print("Chiffre prédit :", chiffre_predit)

plt.imshow(X_test_data[i], cmap='Greys')
plt.show()
```

On rappelle que `Y_predict[i]` est une liste de 10 nombres correspondant à la probabilité de chaque chiffre. Le chiffre prédit s'obtient en prenant le rang de la probabilité maximale, c'est exactement ce que fait la fonction `argmax` de *numpy*.

4.2. Un exercice pour pratiquer

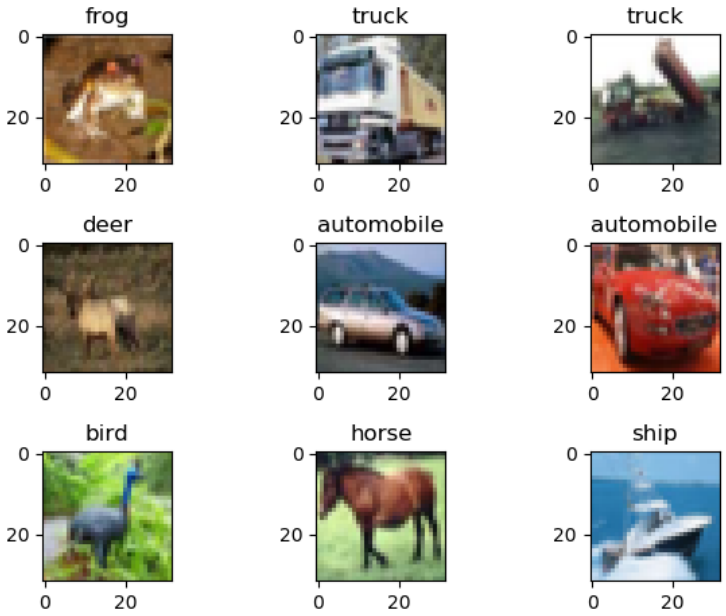
1. **Expérimentez le réseau précédent en modifiant son architecture :**
   - Modifiez le nombre de couches et de neurones.
   - Explorez différentes fonctions d'activation.
2. **Réalisez une analyse d'erreur :**
   - Identifiez des exemples mal classés par le modèle :
     Que font les lignes de code suivantes ?
     ```
     predictions = model.predict(X_test)
     predicted_labels = np.argmax(predictions, axis=1)
     true_labels = np.argmax(Y_test, axis=1)
     ```
   - On utilise ensuite la méthode where de numpy comme ci-dessous :
     `misclassified_indices = np.where(predicted_labels != true_labels)[0]`
     Que retrouve-t-on dans `misclassified_indices` ?
   - Le code ci-dessous afficher les 5 premiers exemples mal classés
     ```
     for idx in misclassified_indices[:5]:
         plt.imshow(X_test[idx].reshape(28, 28), cmap='gray')
         plt.title(f'Predicted: {predicted_labels[idx]}, True: {true_labels[idx]}')
         plt.show()
     ```
     Auriez-vous aussi fait une erreur ?
     Modifiez à nouveau la structure du réseau afin de voir si l'on peut améliorer la prédiction de ces 5 images.

5. Limites de la reconnaissance d'images avec des réseaux denses

On termine par un constat d'échec (provisoire), nos réseaux de neurones atteignent leurs limites lorsque le problème posé se complique. Nous souhaitons reconnaître des petites images et les classer selon 10 catégories.

5.1. Données

La base CIFAR-10 contient 60 000 petites images de 10 types différents.



Plus en détails :

- Il y a 50 000 images pour l'apprentissage et 10 000 pour les tests.
- Chaque image est de taille 32×32 pixels en couleur. Un pixel couleur est codé par trois entiers (r, g, b) compris entre 0 et 255. Une image est donc composée de $32 \times 32 \times 3$ nombres.
- Chaque image appartient à une des dix catégories suivantes : avion, auto, oiseau, chat, biche, chien, grenouille, cheval, bateau et camion.

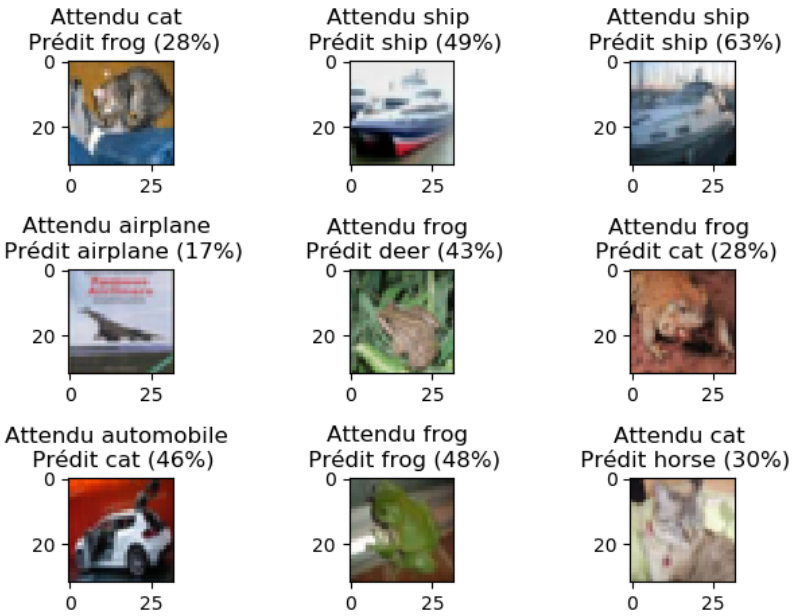
5.2. Programme

```
# Partie A. Données

from tensorflow.keras.datasets import cifar10

(X_train_data,Y_train_data),(X_test_data,Y_test_data)=cifar10.load_data()

num_classes = 10
labels =  ['airplane','automobile','bird','cat','deer',
           'dog','frog','horse','ship','truck']

Y_train = keras.utils.to_categorical(Y_train_data, num_classes)
X_train = X_train_data.reshape(50000,32*32*3)
X_train = X_train.astype('float32')
```

```
X_train = X_train/255

# Partie B. Réseau

modele = Sequential()

p = 30
modele.add(Dense(p, input_dim=32*32*3, activation='sigmoid'))
modele.add(Dense(p, activation='sigmoid'))
modele.add(Dense(p, activation='sigmoid'))
modele.add(Dense(p, activation='sigmoid'))
modele.add(Dense(10, activation='softmax'))

modele.compile(loss='categorical_crossentropy',
               optimizer='adam', metrics=['accuracy'])

modele.summary()

# Partie C. Apprentissage

modele.fit(X_train, Y_train, epochs=10, batch_size=32)
```

5.3. Résultats

Dans le programme ci-dessus avec $p = 30$, il y a 100 000 poids à calculer. Avec beaucoup de calculs (par exemple avec 50 époques et la descente « adam »), on obtient moins de 50% de précision. Ce qui fait que le sujet d'une image sur deux est mal prédit.



Il faut donc une nouvelle approche pour reconnaître correctement ces images !