Reconnaissance d'images avec des réseaux denses 1/2

Objectifs

- *Codage d'une image simple en niveaux de gris et de son label,*
- *Introduction de la fonction d'activation softmax et de l'entropie croisée catégorielle,*
- *Généralités sur la descente de gradient stochastique, parallélisation.*

1. Images et labélisation

Cette partie est une préparation au TP de reconnaissance d'images par un réseau dense, dont l'idée générale est la suivante : nous disposons d'une base de données d'images, ces images sont associées à un label ce label étant un descriptif élémentaire de l'image. Le but sera d'entrainé un réseau dense (à partir de ces images et de leurs labels), afin qu'il soit capable pour une nouvelle image de produire un label et donc un descriptif élémentaire de cette image.
Pour introduire les notions nous nous baserons volontairement sur des images en niveau de gris triviales (de taille $2\times$ 2 pixels). Une image en nuances de gris peut être représentée comme une matrice de pixels où chaque pixel a une intensité comprise entre 0 (noir) et 255 (blanc). Par exemple :

$$\text{Image brute } = \begin{bmatrix} 0 & 255 \\ 127 & 127 \end{bmatrix}.$$

Pour faciliter le traitement, en particulier éviter la saturation des neurones et améliorer l'efficacité de la descente de gradient, les valeurs des pixels sont normalisées entre 0 et 1 lors de l'entraînement du réseau de neurones (chaque valeur est simplement divisée par 255).

$$\text{Image normalisée } = \begin{bmatrix} 0 & 1 \\ 0.5 & 0.5 \end{bmatrix}.$$

De plus, avant d'envoyer l'image dans un réseau dense, on l'aplatit afin de la transformer en un vecteur.

$$\text{Image aplatie } = [0, 1, 0.5, 0.5].$$

1.1. Problème jouet

Considérons le problème-jouet suivant : nous souhaitons entraîner un réseau dense capable de déterminer, à partir d'une image de 2×2 pixels, si une diagonale apparaît dans l'image ou non.
Par exemple l'image normalisée

$$\begin{bmatrix} 0 & 0.8 \\ 1 & 0 \end{bmatrix},$$

que l'on peut afficher avec le code python suivant, présente une diagonale.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import get_cmap
```

```
# Création d'une image 2x2
image = np.array([[0, 1],
                  [1, 0]])

# Affichage
cmp = get_cmap('gray')
plt.imshow(image, cmap=cmp, vmin=0, vmax=1)
plt.show()

# Pour aplatir l'image on utilise la méthode flatten
image_flattened=image.flatten()
```

On veut donc associer le label *diagonale* à cette image. Ce label est un vecteur de dimension 2, par exemple $(1,0)$. La dimension 2 vient du fait que nous cherchons à distinguer deux classes : les images présentant une diagonale et celles qui n'en présentent pas. Le 1 dans la première coordonnée peut être interprété comme : « avec probabilité 1, l'image présente une diagonale » (et donc, avec probabilité 0, elle ne présente pas de diagonale). On peut donc voir un label comme une distribution de probabilités, nous la noterons $\mathbf{p} = (\mathbf{p_1}, \mathbf{p_2})$ dans la suite, ici on donc $p = (1,0)$ le label de notre image.

Ainsi nous obtenons la donnée : image et label

$$\left(\begin{bmatrix} 0 & 1 \\ 0.5 & 0.5 \end{bmatrix}; (1,0)\right).$$

De la même façon

$$\left(\begin{bmatrix} 0.8 & 1 \\ 0 & 0 \end{bmatrix}; (0,1)\right)$$

constituerait, par exemple, une seconde donnée. L'ensemble des couples de données de ce type forme les données d'entrainement du réseau.
Regardons maintenant comment un réseau pourrait produire un label à partir d'une image pour qu'une fois entrainé il puisse produire le bon label pour cette image.
Un réseau dense prend en entrée un vecteur plutôt qu'une matrice, ainsi pour une image 2*2 pixels le réseau dense aura donc 4 entrée (4 neurones) et 2 sorties (les deux coordonnées du label), ainsi qu'un certain nombre de couches cachées. Dans le paragraphe suivant nous voyons comment créer des labels (vecteurs stochastiques) dans un réseau dense, puis comment comparer ces labels créé par le réseau aux labels réels.

2. Softmax et entropie croisée catégorielle

Le label d'une donnée (une image par exemple) est créé à partir des sorties de la dernière couche de neurones en considérant une fonction d'activation un peu particulière appelée softmax.

2.1. La fonction d'activation softmax

Considérons, le cas général pour le moment où la dernière couche est constituée de k neurones (donc k classes), et notons $x = (x_1, \ldots, x_k)$ les k valeurs en sortie de ces neurones avant l'application d'une fonction d'activation (ces valeurs sont appelées " **score**"). La fonction $softmax$ est définie pour tout $i \leqslant k$ par :

$$softmax(x)_i = \frac{e^{x_i}}{e^{x_1} + e^{x_2} + \cdots + e^{x_k}}.$$

On remarque que les $softmax(x)_i$ sont toutes comprises entre 0 et 1 et que leur somme vaut 1, ainsi $softmax(x)_i$ est simplement une probabilité on la notera q_i.
L'ensemble des valeurs $\mathbf{q} = (\mathbf{q_i} = \mathbf{softmax(x)_i}, \mathbf{i} \leqslant \mathbf{k})$ est alors une distribution (ou loi) de probabilités, elles sont prédites par le réseau.

Exercice

1. Afficher l'image suivante en niveau de gris 2×2 :
$$\text{Image } = \begin{bmatrix} 0.9 & 0.2 \\ 0.1 & 0.8 \end{bmatrix},$$
aplatir la matrice en un vecteur :
$$\text{Vecteur aplati } =$$
2. Supposons que le réseau associe les scores suivants à cette image : $x = (1.2, 0.5)$, calculer les probabilités associées $q = (q_1, q_2)$ avec la fonction softmax. En déduire la classe prédite par le réseau.
3. Quelle est la sortie $p = (p_1, p_2)$ attendue pour l'image de la question 1 ?

L'étape suivante consiste à comparer les deux distributions q et p. En effet, une fois le réseau entraîné, nous souhaitons qu'il produise une distribution q aussi proche que possible de p.

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 probabilités prédite par un modèle de réseau de neurones et la distribution de probabilités réelle.
La categorical crossentropy (entropie croisée catégorielle : ECC) est une fonction de perte couramment utilisée en apprentissage automatique, notamment pour les tâches de classification multiclasse. Plus précisément, elle mesure 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 $q_i = softmax(x)_i$) 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

0. Calculer l'entropie croisée des distribution p et q, de l'exercice précédent. Que dire du réseau qui a produit q ?

Soit les motifs simples 2×2 et ses labels :

- Image 1 : $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, label $p_{Image\ 1} = [1, 0]$.
- Image 2 : $\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$, label $p_{Image\ 2} = [0, 1]$.

1. Supposons que les scores produits par le réseau sont :
$$x_{\text{Image 1}} = [1.5, 0.2], \quad x_{\text{Image 2}} = [0.3, 1.1].$$
2. Calculer les probabilités q pour chaque cas.
3. Utiliser l'ECC pour comparer les probabilités aux labels attendus p.
4. Conclure sur les performances du réseau dans la distinction entre ces motifs.

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

La descente de gradient stochastique (abrégée en *SGD*) est une méthode d'optimisation qui permet d'accélérer le calcul de la descente de gradient lorsqu'une fonction de perte dépend d'un grand nombre de données. Au lieu de calculer le gradient complet, puis de mettre à jour le paramètre en utilisant l'ensemble des données, on calcule un gradient *local* et une mise à jour pour chaque donnée. Ce processus doit naturellement être répété pour toutes les données. Le terme *stochastique* provient du fait que les données sont mélangées de manière aléatoire avant le calcul des gradients. Une alternative consiste à regrouper les données en *lots* (*batches*) et à calculer un gradient pour chaque lot. Cette approche présente plusieurs avantages : elle réduit partiellement le bruit inhérent aux mises à jour individuelles et permet une parallélisation plus efficace du calcul des gradients.
Rappelons que l'entrainement d'un réseau de neurones consiste à la minimisation d'une fonction d'erreur notée E ici. Cette erreur est elle même composée d'une somme d'erreurs locales notées E_i, ainsi l'erreur global est donnée par $E = \sum_{i=1}^{N} E_i$ où N est le nombre de données disponibles pour l'entrainement. En fonction du type de problème, nous avons à faire à plusieurs forment pour ces erreurs. Jusque là nous en avons vu deux : l'erreur des moindres carrés ($E_i = (y_i - F(x_i))^2$ où $F(x_i)$ est la sortie du réseau et dépend donc des paramètres de celui-ci, et l'entropie croisée ($E_i = p_i \log q_i$) de même q_i étant la sortie produite par le réseau elle dépend aussi des paramètres du réseau.
Nous ne verrons pas ici les détails mathématiques couvrant les propriétés du gradient stochastique (cela peut faire l'objet d'un cours). Cependant nous donnons quelques idées générales, les avantages et inconvénients des différentes méthodes usuelles de descente de gradient.

A. Descente de gradient classique. Supposons que notre réseau de neurones possèdent m paramètres à optimiser. L'optimisation débute par une initialisation aléatoire des m paramètres que nous noterons $\omega^0 = (\omega_1^0, \ldots, \omega_m^0) \in \mathbb{R}^m$, puis on applique la formule de récurrence :
$$\omega^{k+1} = \omega^k - \delta \nabla E(\omega^k).$$
Pour appliquer cette formule, il faut , bien entendu, calculer chaque gradients locaux $\text{grad}\, E_i(\omega_k)$, les garder en mémoire, puis les sommer pour finalement obtenir
$$\nabla E(\omega^k) = \sum_{i=1}^{N} \nabla E_i(\omega^k).$$
Le calcul d'un gradient complet sur l'ensemble des données constitue l'un des principaux inconvénients de la descente de gradient classique, en effet à la fois m et N sont grands. Par exemple m peut atteindre

la centaine de milliards pour certains modèles, et plusieurs centaines de milliers dans notre problème de reconnaissance d'images et N le nombre de données doit lui aussi être suffisamment grand pour pouvoir optimiser ces m paramètres. Cette méthode présente toutefois l'avantage d'être très peu bruitée, puisque le gradient est estimé à partir de l'ensemble des données, ce qui atténue fortement les fluctuations aléatoires. Un second inconvénient majeur est qu'elle se parallélise difficilement. En effet, même si le calcul des gradients élémentaires est indépendant d'un exemple à l'autre, la descente de gradient classique impose de traiter l'intégralité des données avant de pouvoir effectuer une mise à jour. Cela entraîne un coût mémoire important, une faible fréquence de mise à jour (une seule mise à jour toutes les N données) et des synchronisations lourdes, ce qui limite fortement l'efficacité sur les architectures parallèles comme les GPU.

B. Descente de gradient stochastique (SGD).
Pour diminuer la quantité de calculs à chaque itération, l'idée de la descente de gradient stochastique est de ne considérer qu'un seul terme d'erreur E_i à la place de l'erreur totale $E = \sum_{j=1}^{N} E_j$. Autrement dit, à l'itération k, on met à jour les paramètres selon la formule

$$\omega^{k+1} = \omega^k - \delta \nabla E_i(\omega^k),$$

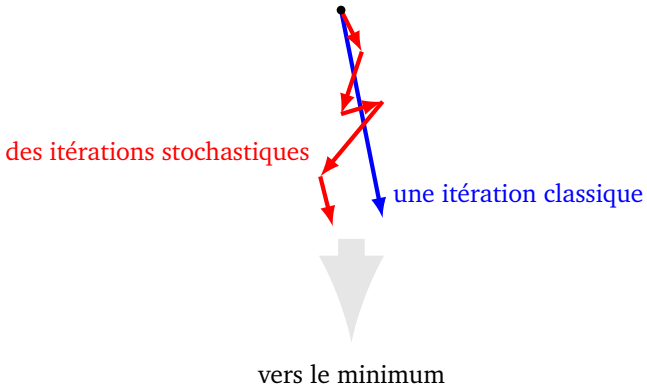
où E_i est l'erreur locale associée à une donnée particulière.

Randomisation des données. En pratique, on ne parcourt pas les données dans l'ordre $1, 2, \ldots, N$. Avant chaque passage complet sur les N données (appelé *époque*), on commence par mélanger aléatoirement les données. Ainsi, au lieu d'utiliser successivement les erreurs $E_1, E_2, \ldots, E_N$, on utilise

$$E_{i_1}, E_{i_2}, \ldots, E_{i_N},$$

où $(i_1, \ldots, i_N)$ est une permutation aléatoire de $\{1, \ldots, N\}$. Ce mélange évite que l'ordre des données influence l'entraînement et renforce le caractère stochastique de l'algorithme.

Quel est l'intérêt de cette méthode ? Dans la descente de gradient classique, chaque itération nécessite le calcul d'un « gros » gradient impliquant l'ensemble des N données. Dans la descente de gradient stochastique, au contraire, on calcule successivement N « petits » gradients (un pour chaque donnée) et l'on met à jour beaucoup plus fréquemment les paramètres. Même si chaque mise à jour est plus bruitée, l'ensemble progresse efficacement vers un minimum.



Voici les premières itérations de cet algorithme : Avant de commencer une époque, on génère une permutation aléatoire $(i_1, \ldots, i_N)$.

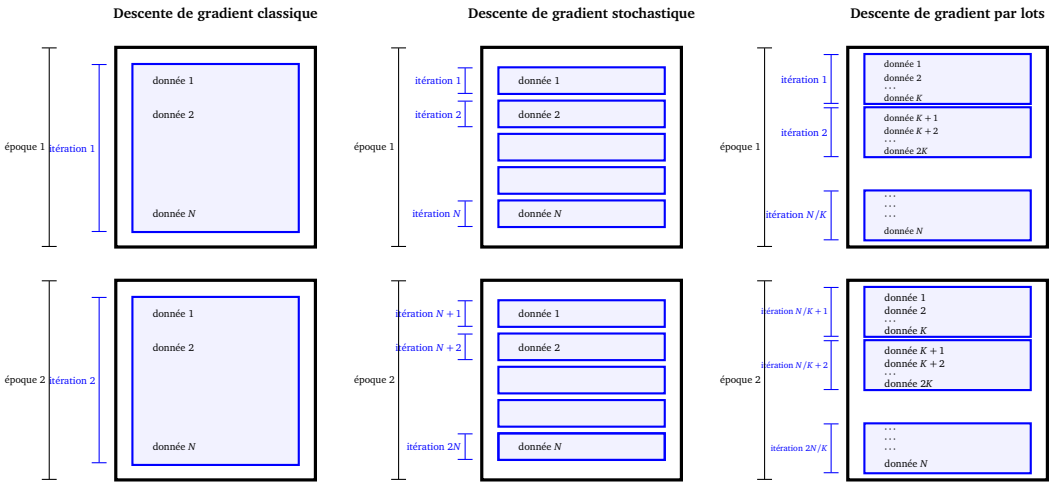
- On part d'un point initial ω^0.
- On calcule
$$\omega^1 = \omega^0 - \delta \nabla E_{i_1}(\omega^0).$$
- Puis
$$\omega^2 = \omega^1 - \delta \nabla E_{i_2}(\omega^1), \qquad \omega^3 = \omega^2 - \delta \nabla E_{i_3}(\omega^2),$$
et ainsi de suite.

- Après N mises à jour, on a utilisé toutes les données une fois :
$$\omega^N = \omega^{N-1} - \delta \nabla E_{i_N}(\omega^{N-1}).$$
- On commence alors une nouvelle époque : on mélange de nouveau les données et on recommence avec une nouvelle permutation.

L'algorithme est arrêté soit après un nombre d'itérations fixé à l'avance, soit lorsque ω^N est jugé suffisamment proche d'un minimum !

Avantages et inconvénients de la SGD simple.
La descente de gradient stochastique présente un avantage majeur : son coût par itération est très faible, puisqu'elle ne nécessite que le calcul du gradient d'une seule erreur locale. Cela permet des mises à jour beaucoup plus fréquentes et une exploration plus dynamique du paysage d'erreur. De plus, le caractère stochastique des gradients peut aider l'algorithme à s'extraire de minima locaux ou de zones où le gradient est presque nul (plateaux).
Cependant, cette méthode introduit du bruit dans les mises à jour. Autrement dit, la direction moyenne suivie est correcte, mais chaque mise à jour est perturbée par une variance potentiellement élevée si les gradients locaux diffèrent beaucoup les uns des autres. Cette variance se manifeste par des oscillations autour du minimum, ce qui ralentit la convergence et impose souvent un choix plus petit pour le taux d'apprentissage. Un autre inconvénient de la SGD « pure » (avec une seule donnée par itération) est qu'elle se prête très mal à la parallélisation. À chaque étape, l'algorithme a besoin du point ω^k pour calculer le gradient local $\nabla E_{i_k}(\omega^k)$, puis de ce gradient pour former ω^{k+1}. La mise à jour dépend donc entièrement de la mise à jour précédente : l'algorithme est strictement séquentiel. Il est impossible, par exemple, de calculer simultanément plusieurs gradients locaux pour avancer plus vite, puisqu'on ne peut pas connaître ω^{k+1} tant que l'on n'a pas d'abord calculé ω^k.
Cette structure séquentielle rend la SGD simple incapable d'exploiter efficacement l'architecture parallèle d'un GPU, qui est conçue pour traiter plusieurs données en même temps. Autrement dit, même si l'on disposait d'un grand nombre de cœurs de calcul, chaque itération de la SGD simple n'utiliserait qu'une fraction infime de la puissance disponible. C'est l'une des raisons majeures pour lesquelles, en pratique, on préfère utiliser des méthodes par *mini-batch*, qui permettent de paralléliser le calcul de plusieurs gradients simultanément.

C. Descente de gradient par lots (ou *mini-batch*).
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 par itération).
La descente de gradient par ***lots*** (ou ***mini-lots***, *mini-batch*) consiste à diviser les données en paquets de taille K. Pour chaque paquet (appelé « lot »), on calcule un gradient et on effectue une mise à jour.
Après N/K itérations, on a parcouru l'ensemble du jeu de données : on parle alors d'une ***époque***.



La mise à jour est donnée par

$$\omega^{k+1} = \omega^k - \delta \nabla\big(E_{j_0+1} + E_{j_0+2} + \cdots + E_{j_0+K}\big)(\omega^k),$$

c'est-à-dire le gradient de la somme des erreurs du lot courant. Pour l'itération suivante, on repart de ω^{k+1} et l'on utilise les K données suivantes :

$$\omega^{k+2} = \omega^{k+1} - \delta \nabla\big(E_{j_0+K+1} + \cdots + E_{j_0+2K}\big)(\omega^{k+1}).$$

Remarque.

- Pour $K = 1$, on retrouve exactement la descente de gradient stochastique. Pour $K = N$, on retrouve la descente de gradient classique.
- Cette méthode combine le meilleur des deux mondes : elle réduit le bruit des gradients (contrairement à la SGD pure), tout en permettant des mises à jour fréquentes et moins coûteuses qu'avec la descente de gradient classique.
- Elle permet d'exploiter la parallélisation : pour un lot de taille K, on peut calculer les gradients locaux ∇E_i en parallèle sur K processeurs (ou sur un GPU), puis les additionner pour obtenir le gradient global du lot.
- Comme pour la SGD, il est d'usage de mélanger aléatoirement les données avant chaque époque.

Exercice

Répondre **Vrai** ou **Faux** aux affirmations suivantes.

1. Dans la descente de gradient classique, chaque mise à jour nécessite le calcul du gradient sur l'ensemble des N données.
2. La descente de gradient stochastique (SGD) introduit du bruit dans les mises à jour car les gradients sont calculés à partir d'une seule donnée.
3. Le mélange aléatoire des données avant chaque époque limite l'influence de leur ordre sur l'apprentissage.
4. La descente de gradient classique est très facile à paralléliser car les mises à jour des paramètres peuvent être effectuées indépendamment du calcul des gradients.
5. La méthode par mini-batch permet de paralléliser le calcul des gradients locaux pour un lot donné.
6. Pour une taille de lot $K = N$, la méthode mini-batch devient équivalente à la descente de gradient stochastique (SGD).
7. La SGD (avec $K = 1$) permet de calculer plusieurs mises à jour de paramètres en parallèle pour un même réseau, sans dépendance séquentielle entre elles.
8. Les mini-batches permettent de réduire la variance des gradients par rapport à la SGD pure tout en gardant des mises à jour plus fréquentes que dans la descente de gradient classique.