Les bases de l'apprentissage profond

Quentin Torroba

15 novembre 2023

Table des matières

Table des figures

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

1 Le perceptron

Il est fort probable que vous ayez déjà rencontré cette représentation d'un réseau de neurones :

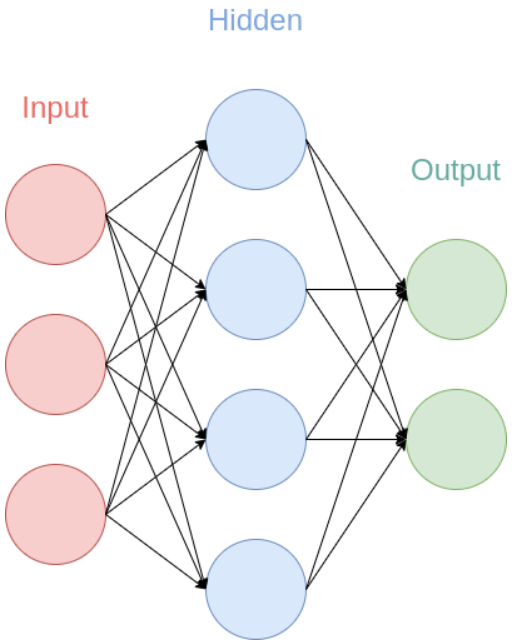


Figure 1: Représentation classique d'un réseau de neurones

On retrouve alors trois parties importantes:

- La couche d'entrée, également appelée "input layer", joue, à l'instar d'un algorithme classique, le rôle de représenter les données d'entrée. Dans le cas d'un classificateur d'images, par exemple, lorsque nous cherchons à déterminer si une image représente un chat ou non, les pixels de l'image constituent les données d'entrée. (Pour une image en noir et blanc de 28 pixels sur 28, nous aurions 784 entrées.)
- La couche de sortie, ou "output layer", représente les données de sortie. Dans notre exemple précédent, nous aurions une seule sortie : la probabilité que l'image représente un chat ou non.
- La ou les couche(s) cachée(s), ou "hidden layer(s)", constitue(nt) une caractéristique propre aux réseaux de neurones. Le nombre d'éléments par couche et le nombre de couches dépendent de l'architecture spécifique de notre réseau de neurones. Ces couches sont composées d'éléments appelés perceptrons.

Le perceptron est l'un des éléments fondamentaux de l'apprentissage automatique et des réseaux de neurones artificiels. Il s'agit d'un modèle mathématique inspiré par le fonctionnement des neurones dans le cerveau humain, utilisé pour effectuer des tâches de classification et d'approximation de fonctions.

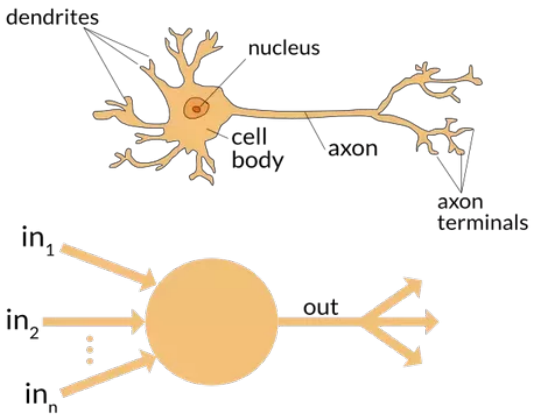


Figure 2: Un neurone et son approximation

Le perceptron consiste en une unité computationnelle élémentaire qui prend un ensemble de valeurs d'entrée pondérées, les additionne, puis applique une fonction d'activation pour générer une sortie. Voici les étapes de fonctionnement du perceptron :

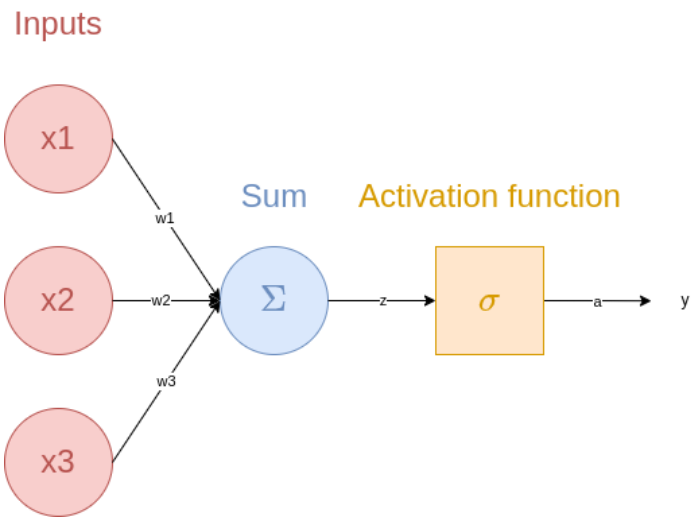


Figure 3: Le perceptron

1. **Poids** et **Biais** : Chaque entrée est associée à un poids qui représente l'importance relative de cette entrée pour la sortie du perceptron. Un biais est rajouté afin d'ajuster le point de départ, dans le cas d'une régression linéaire, on peut le voir comme l'ordonnée à l'origine, et il peut être modéliser comme une entrée de plus, dont la valeur est toujours 1, et le poids est égal à b

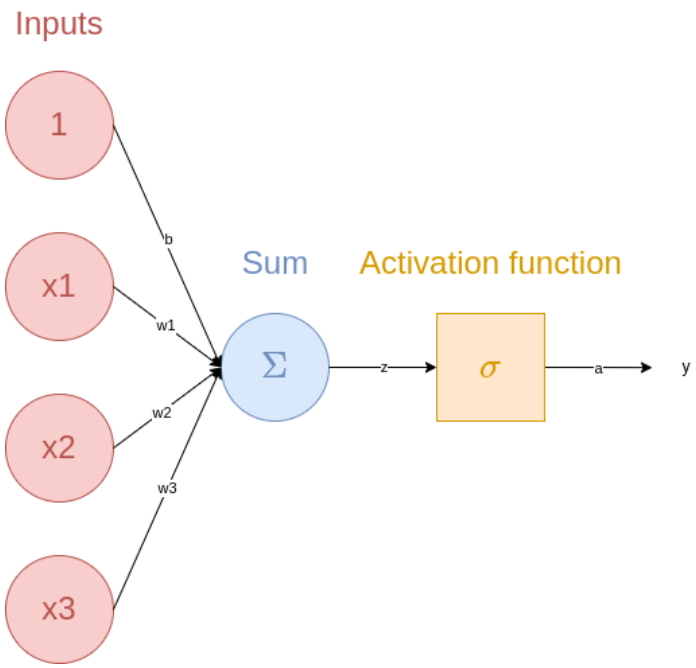


Figure 4: Réseau avec bias

2. **Combinaison linéaire** : Les valeurs d'entrée sont multipliées par leurs poids correspondants et sommées, puis le biais est ajouté à cette somme pondérée. Cela crée une combinaison linéaire des entrées pondérées. Cette combinaison linéaire peut être vue comme une multiplication de matrices.

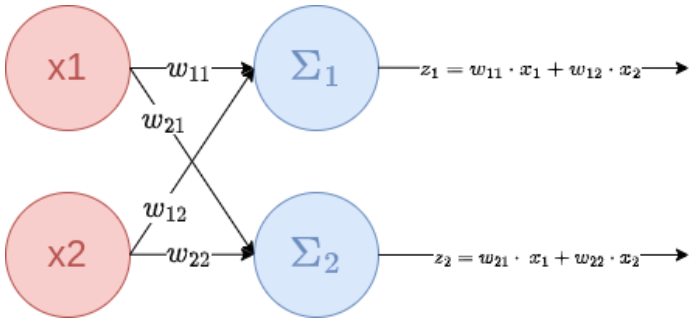


Figure 5: Poids pour 2 entrées et 2 perceptrons

$$z = W \cdot x = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} w_{11} \cdot x_1 + w_{12} \cdot x_2 \\ w_{21} \cdot x_1 + w_{22} \cdot x_2 \end{bmatrix}$$

3. **Fonction d'activation** : La combinaison linéaire est ensuite introduite dans une fonction d'activation. Cette fonction non linéaire introduit de la complexité et de la non-linéarité dans le modèle. Nous étudierons plus tard l'importance de cette non-linéarité

Afin de mieux le comprendre, nous allons maintenant étudier le perceptron en créant un réseau de neurones avec une entrée, une sortie, et une couche cachée avec un seul perceptron.

Sachant qu'un perceptron pour une entrée aura un poids et un biais, il peut alors approximer une fonction linéaire, sa sortie étant $z = w \cdot x + b$

On suppose que nous voulons approximer la fonction $f(x) = 2x + 1$ à l'aide du perceptron. Notre perceptron prendra une seule entrée x et donnera une sortie approximée.

1.1 Création du perceptron

Les paramètres du perceptron sont dans un premier temps aléatoire

```
w = np.random.randn()
b = np.random.randn()
print(f"Poids: {w:.4f}, Biais: {b:.4f}")
```

```
Poids: 0.4967, Biais: -0.1383
```

1.2 Fonction d'activation

Pour cette approximation linéaire, nous n'avons pas besoin d'une fonction d'activation non linéaire. Nous utiliserons simplement une fonction d'identité qui renvoie la même valeur.

```
def activation_function(x):
    return x
```

1.3 Calcul de la sortie du perceptron

Maintenant, implémentons le calcul de la sortie du perceptron en combinant linéairement l'entréee avec les poids et le biais, puis en appliquant la fonction d'activation.

```
def perceptron_output(x):
    linear_combination = w * x + b
    output = activation_function(linear_combination)
    return output
```

Nous observons alors qu'un perceptron à une entrée permet d'effectuer une régression linéaire, nous allons alors étudier cet exemple pour créer notre premier algorithme d'apprentissage profond.

1.4 Approximation de la fonction linéaire

Nous allons maintenant générer des valeurs d'entrée, calculer les sorties du perceptron et les comparer aux valeurs réelles de la fonction $f(x) = 2x + 1$.

```
# Génération de valeurs d'entrée
np.random.seed(42)
x_values = np.random.rand(2, 1)

# Calcul des sorties du perceptron
perceptron_outputs = [perceptron_output(x) for x in x_values]

# Calcul des valeurs réelles de la fonction
true_outputs = 2 * x_values + 1

# Tracé de la fonction réelle et de l'approximation du perceptron
plt.figure(figsize=(8, 6))
plt.scatter(x_values, true_outputs, label='Vraie fonction : 2x + 1')
plt.plot(x_values, perceptron_outputs, color='red' , label='Perceptron')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Approximation de la fonction linéaire avec un Perceptron')
plt.legend()
plt.grid(True)
plt.show()
```

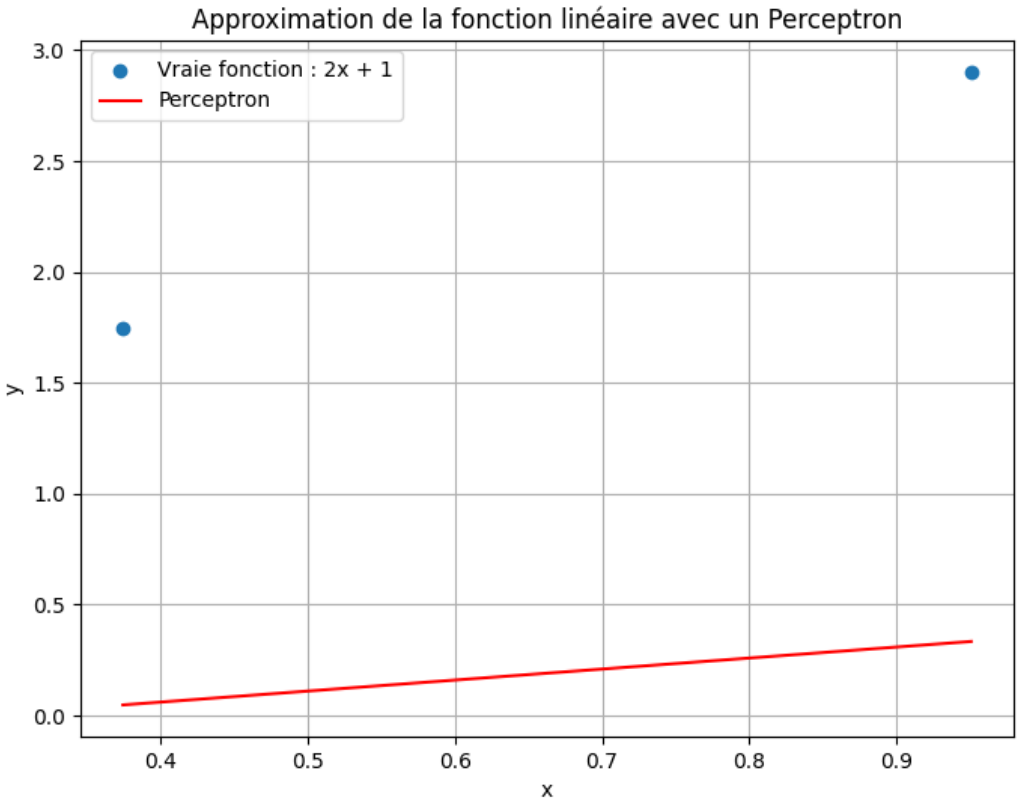


Figure 6: Approximation de la fonction linéaire avec un Perceptron

2 Choix de fonction de perte et de coût

Comme nous pouvons l'observer, le perceptron n'a pas réussi à approximer la fonction du premier coup. Nous devons alors quantifier à quel point les prédictions du modèle sont éloignées des valeurs réelles, et nous devons déterminer l'influence des paramètres poids et biais sur l'erreur, afin de les corriger. Pour cela, nous utilisons une fonction de perte (ou fonction de coût) qui mesure cette différence. Dans le cas du problème de régression, où nous essayons d'approximer une fonction, une fonction de perte couramment utilisée est l'erreur quadratique moyenne (MSE - Mean Squared Error).

La fonction MSE est définie comme suit, où n est le nombre d'exemples d'entraînement, y_i sont les valeurs réelles et $\hat{y}_i$ sont les prédictions du modèle : $MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

Pour faciliter les calculs, notre fonction de coût sera définie comme $\mathcal{C} = \frac{MSE}{2}$.

Pour comprendre ce choix de fonction de coût, nous pouvons la visualiser :

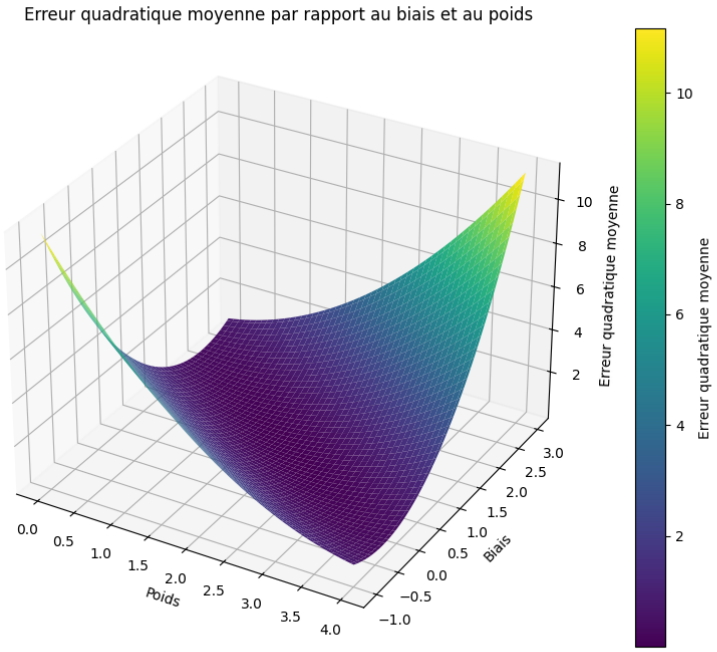


Figure 7: Erreur quadratique moyenne par rapport au biais et au poids

En particulier, uniquement par rapport au poids:

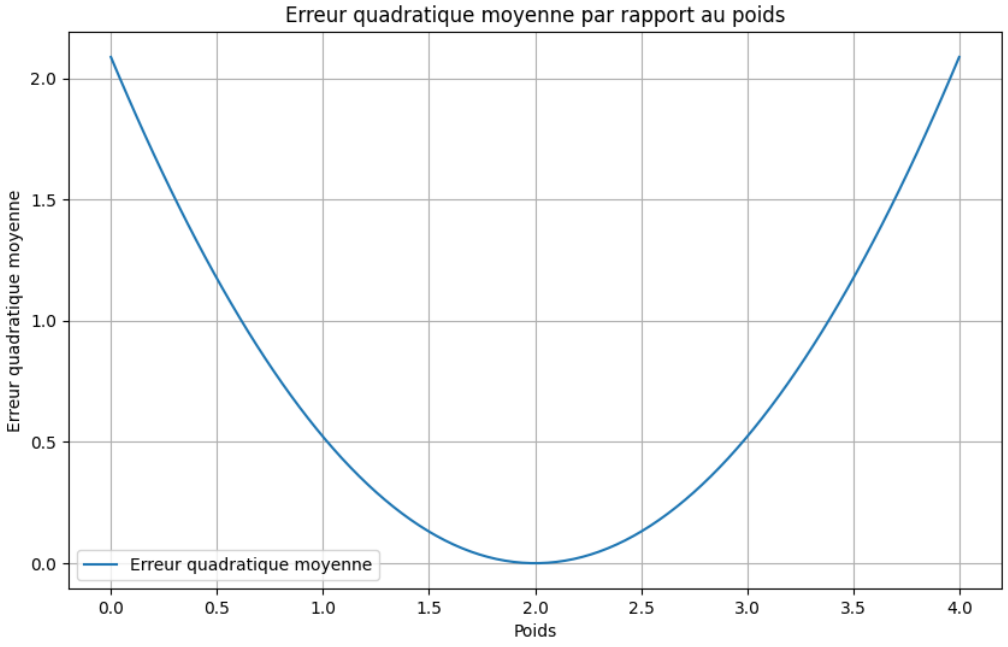


Figure 8: Erreur quadratique moyenne par rapport au poids

3 Rétropropagation et Descente de gradient

Cette fonction est alors idéale pour minimiser l'erreur par rapport aux paramètres. En étudiant le gradient, qui nous indique la direction dans laquelle la fonction de perte augmente, nous pouvons la minimiser en suivant la direction opposée du gradient.

Ceci constitue la base de l'algorithme de descente de gradient.

3.1 Calcul de gradient

Pour calculer les gradients, nous effectuons un simple règle de chaîne:

$$\frac{\partial \mathcal{C}}{\partial w} = \frac{\partial \mathcal{C}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w}$$

et

$$\frac{\partial \mathcal{C}}{\partial b} = \frac{\partial \mathcal{C}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b}$$

Sachant que $\mathcal{C} = \frac{1}{2n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$ et $\hat{y} = wx + b$ avec x l'entrée, on a alors

$$\frac{\partial \mathcal{C}}{\partial w} = \frac{1}{2n} 2 \sum_{i=1}^{n} (y_i - \hat{y}_i) \cdot x = \frac{x}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)$$

et

$$\frac{\partial \mathcal{C}}{\partial b} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)$$

3.2 Descente de gradient

Après avoir calculé le gradient, on le multiplie par un taux d'apprentissage, ou "learning rate", généralement compris entre 0,001 et 0,1. Une fois qu'un taux d'apprentissage est choisi, les ajustements des paramètres peuvent être effectués en utilisant cette formule simple : `w -= gradient(w) * lr` Ce qui est appelé un pas, ou "step". Ce qui va permettre d'ajuster les paramètres pour minimiser la perte. Il est important de prendre un taux d'apprentissage, ni trop faible, car pouvant prendre trop de pas:

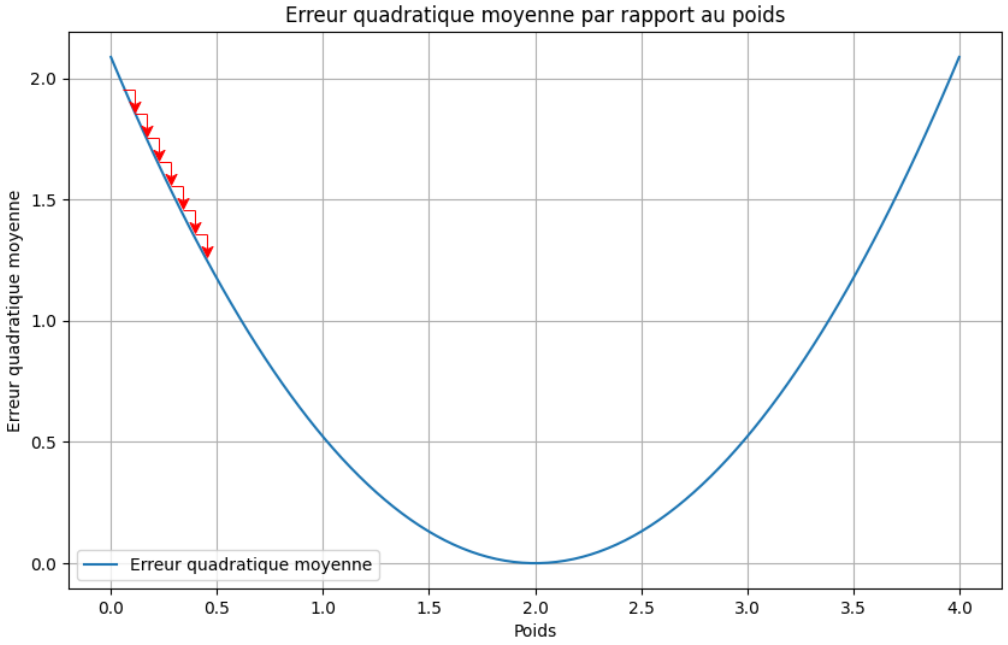


Figure 9: Exemple avec taux d'apprentissage faible

Ni trop élevé, car pouvant empirer la perte.

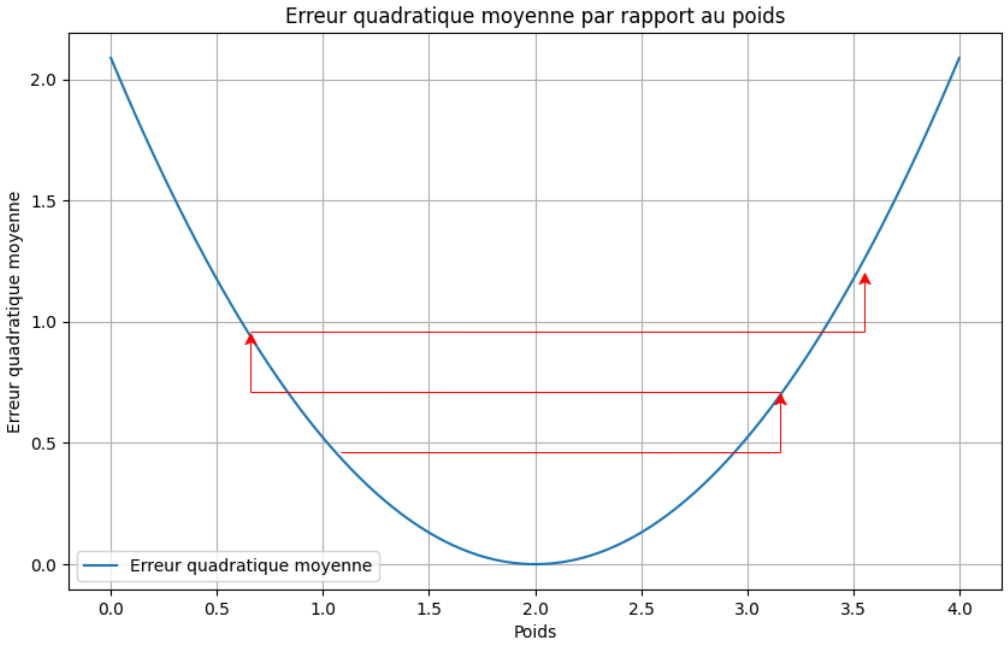


Figure 10: Exemple avec taux d'apprentissage élevé

Maintenant que nous avons exploré la descente de gradient et comment elle nous permet de minimiser l'erreur par rapport aux paramètres, il est temps d'aborder deux concepts importants

dans le processus d'apprentissage en profondeur : les epochs et les lots (batches).

3.3 Epoch

Un epoch est une phase complète d'entraînement au cours de laquelle l'algorithme voit l'ensemble complet de données d'entraînement une fois. Il consiste à parcourir l'ensemble de données du début à la fin, en utilisant la méthode de descente de gradient pour ajuster les paramètres du modèle. Réaliser plusieurs epochs signifie répéter ce processus plusieurs fois, permettant au modèle d'améliorer ses performances à chaque passage.

3.4 Batches

Lors de l'entraînement d'un modèle, il peut être inefficace et exigeant en termes de mémoire de présenter toutes les données d'entraînement à la fois. C'est là que les lots entrent en jeu. Un lot (ou batch en anglais) est un sous-ensemble des données d'entraînement que l'algorithme utilise pour effectuer une mise à jour des paramètres. Plutôt que d'utiliser l'ensemble complet de données en une seule fois, les données sont divisées en plusieurs lots.

L'utilisation de lots présente plusieurs avantages. Elle permet de :

- **Réduire les besoins en mémoire :** Les données peuvent être trop volumineuses pour être chargées en une seule fois, mais elles peuvent être gérées en morceaux plus petits.
- **Accélérer le calcul :** Les opérations sur des lots sont généralement plus rapides que sur l'ensemble complet de données.
- **Converger plus rapidement :** Les mises à jour fréquentes des paramètres grâce aux lots peuvent permettre au modèle de converger plus rapidement vers une solution.

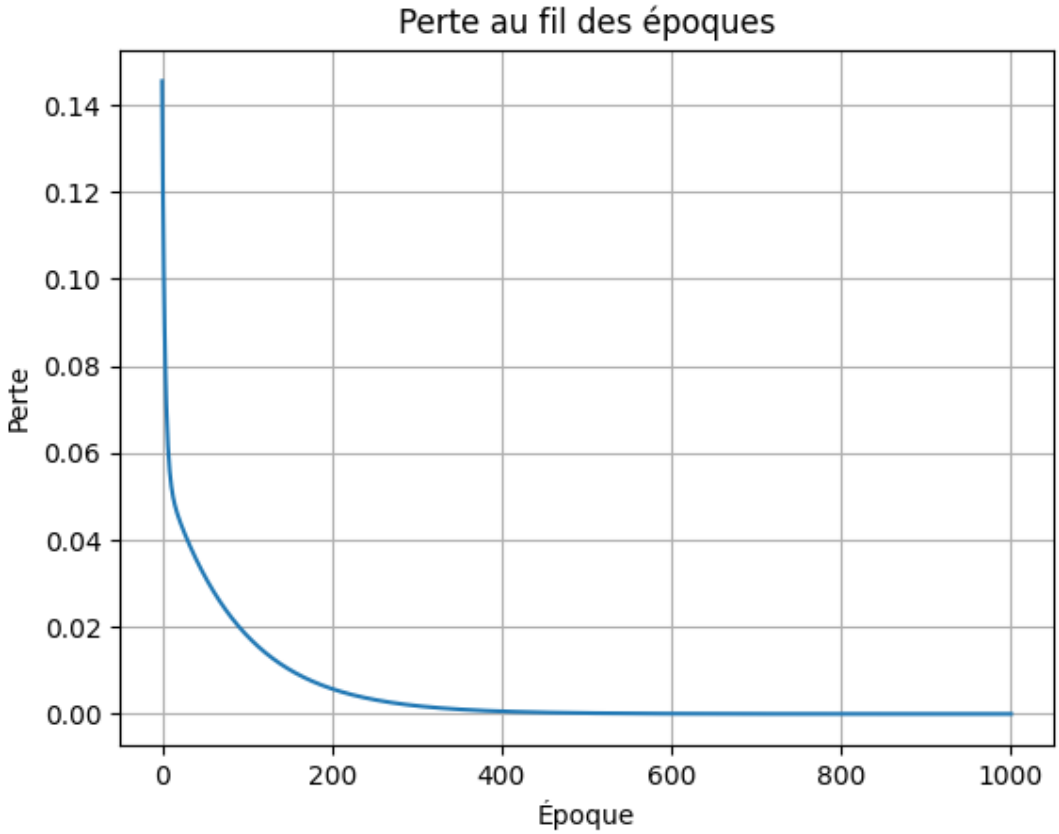
La taille du lot, souvent désignée par "batch size" en anglais, détermine le nombre d'échantillons d'entraînement inclus dans chaque lot. Elle peut varier en fonction des besoins et des contraintes du problème. Des valeurs courantes pour la taille de lot sont 32, 64, 128, etc. Dans notre cas, nous n'utilisons pas de lot.

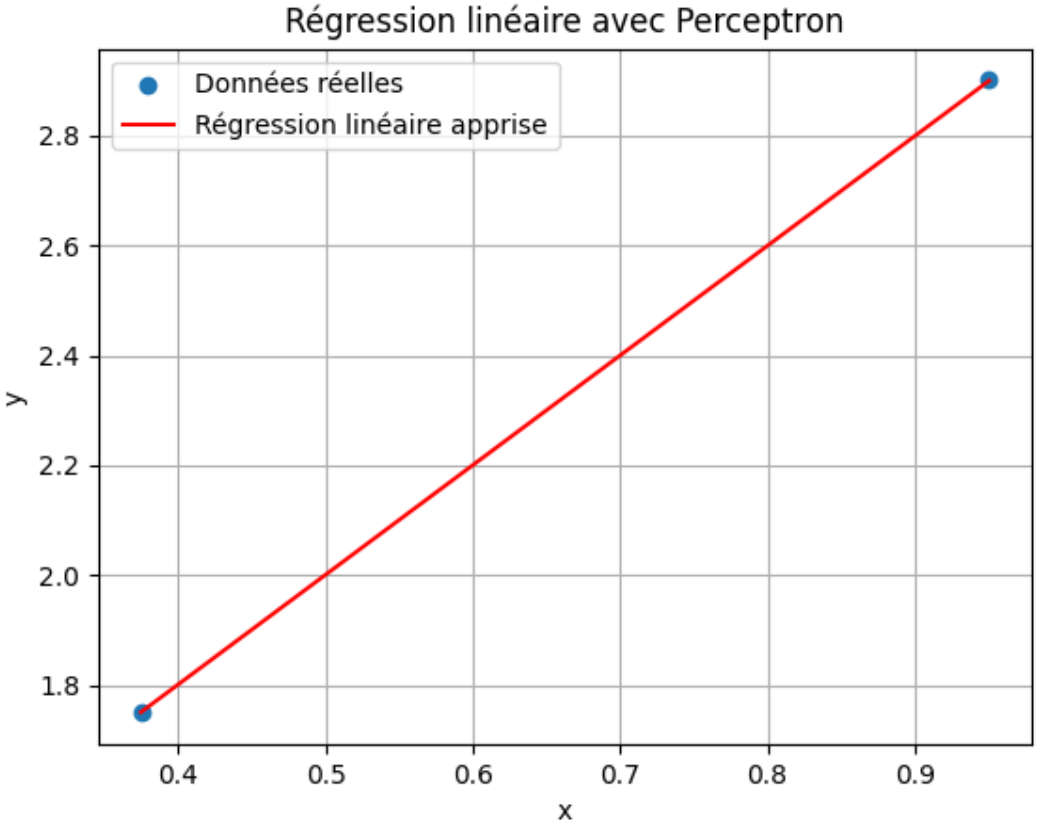
Maintenant que nous comprenons le principe, nous pouvons implémenter en python:

```
# Génération de données synthétiques pour l'entraînement
np.random.seed(42)
X = np.random.rand(2, 1)
y = 2 * X + 1

# Initialisation des paramètres du modèle
w = np.random.randn()  # Poids initial
```

```python
b = np.random.randn()  # Biais initial
lr = 0.1  # Taux d'apprentissage

# Fonction d'activation (identité)
def activation(x):
    return x

# Fonction de coût
def criterion(y_true, y_pred):
    return 1/2 * np.mean((y_true - y_pred)**2)

# Entraînement du modèle
num_epochs = 1001
losses = []

for epoch in range(num_epochs):
    # Calcul des prédictions
    predictions = activation(w * X + b)

    # Calcul de la perte
    loss = criterion(y, predictions)
    losses.append(loss)

    # Calcul des gradients
    dw = (1/len(X)) * np.sum((predictions - y) * X)
    db = (1/len(X)) * np.sum(predictions - y)

    # Mise à jour des paramètres
    w -= lr * dw
    b -= lr * db

    if epoch % 100 == 0:
        print(f'Époque {epoch}, Perte : {loss:.4f}')
print("============================")
print(f'Poids: {w:.4f}, Biais: {b:.4f}')
print("============================")
# Affichage de la perte au fil des époques
plt.plot(losses)
```

```
plt.xlabel('Époque')
plt.ylabel('Perte')
plt.title('Perte au fil des époques')
plt.grid(True)
plt.show()

# Affichage de la régression linéaire apprise
plt.scatter(X, y, label='Données réelles')
plt.plot(X, w * X + b, color='red', label='Régression linéaire apprise')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Régression linéaire avec Perceptron')
plt.legend()
plt.grid(True)
plt.show()

Époque 0, Perte : 0.1455
Époque 100, Perte : 0.0180
Époque 200, Perte : 0.0058
Époque 300, Perte : 0.0019
Époque 400, Perte : 0.0006
Époque 500, Perte : 0.0002
Époque 600, Perte : 0.0001
Époque 700, Perte : 0.0000
Époque 800, Perte : 0.0000
Époque 900, Perte : 0.0000
Époque 1000, Perte : 0.0000
============================
Poids: 1.9961, Biais: 1.0027
============================
```


Perte au fil des époques
Perte
Époque
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
200
400
600
800
1000


Régression linéaire avec Perceptron
Données réelles
Régression linéaire apprise
y
x
2.8
2.6
2.4
2.2
2.0
1.8
0.4
0.5
0.6
0.7
0.8
0.9

4 Plus loin: Réseau de neurones Feed-Forward

Nous avons vu comment un neurone, ou perceptron, fonctionne à l'échelle d'une entrée et d'une sortie. Cependant, nous allons étudier des concepts plus complexes qu'une régression linéaire, impliquant plusieurs entrées et sorties. Pour résoudre ces problèmes, il sera nécessaire de regrouper plusieurs perceptrons en un réseau. Le réseau classique est le réseau Feed-Forward ou à multiples couches de perceptrons (MLP), comme illustré dans la Figure 1. Nous allons maintenant créer notre tout premier MLP dans le but de classifier des images.

Pour cela, nous allons travailler avec le jeu de données MNIST, dont nous pouvons observer un exemple ci-dessous:

```
from sklearn.datasets import fetch_openml

mnist = fetch_openml("mnist_784", parser="auto")

image_data = mnist.data.iloc[0].to_numpy().reshape(28, 28)

plt.subplots(figsize=(2, 2))
plt.imshow(image_data, cmap="gray")
plt.title("Image MNIST")
plt.show()
```

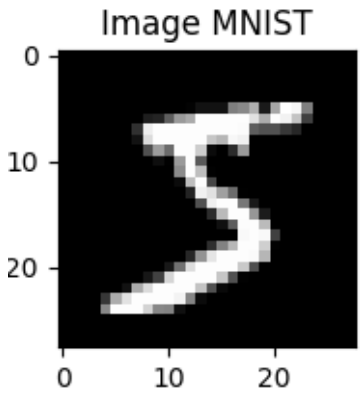


Figure 11: Image MNIST

Ce dataset représente 60 000 images de 28 par 28 pixels en nuances de gris, avec des chiffres allant de 0 à 9. Afin de comprendre comment réaliser un réseau de neurones "naïf" prenant en entrée des images, il suffit de comprendre comment une image est encodée.

```
df = pd.DataFrame(image_data[5:25,7:22])
print(df)
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	0	0	3	18	18	18	126	136	175	26	166	255
1	0	30	36	94	154	170	253	253	253	253	253	225	172	253	242
2	49	238	253	253	253	253	253	253	253	253	251	93	82	82	56
3	18	219	253	253	253	253	253	198	182	247	241	0	0	0	0
4	0	80	156	107	253	253	205	11	0	43	154	0	0	0	0
5	0	0	14	1	154	253	90	0	0	0	0	0	0	0	0
6	0	0	0	0	139	253	190	2	0	0	0	0	0	0	0
7	0	0	0	0	11	190	253	70	0	0	0	0	0	0	0
8	0	0	0	0	0	35	241	225	160	108	1	0	0	0	0
9	0	0	0	0	0	0	81	240	253	253	119	25	0	0	0
10	0	0	0	0	0	0	0	45	186	253	253	150	27	0	0
11	0	0	0	0	0	0	0	0	16	93	252	253	187	0	0
12	0	0	0	0	0	0	0	0	0	0	249	253	249	64	0
13	0	0	0	0	0	0	0	46	130	183	253	253	207	2	0
14	0	0	0	0	0	39	148	229	253	253	253	250	182	0	0
15	0	0	0	24	114	221	253	253	253	253	201	78	0	0	0
16	0	23	66	213	253	253	253	253	198	81	2	0	0	0	0
17	171	219	253	253	253	253	195	80	9	0	0	0	0	0	0
18	253	253	253	253	244	133	11	0	0	0	0	0	0	0	0
19	253	212	135	132	16	0	0	0	0	0	0	0	0	0	0

Toutes les images sont ainsi représentées par 784 valeurs, allant de 0 à 255, correspondant aux 784 pixels en nuances de gris. Nous pouvons alors créer un réseau avec 784 entrées. Puisque nous devons associer chaque image à une valeur de 0 à 9, nous aurons 10 sorties : une pour chaque chiffre.

L'intuition est alors que chaque pixel aura une coefficient de contribution pour chaque chiffre. On peut imaginer que le pixel du milieu (14,14) aura peu d'importance pour le chiffre zéro, et aura alors un coefficient faible pour ce chiffre, mais plus d'importance pour une chiffre comme trois ou cinq, et donc un coefficient plus élevé.

4.1 Mise en place du réseau

4.1.1 Traitement des données

4.1.1.1 Normalisation des données La normalisation des données est une procédure de prétraitement essentielle dans les tâches d'apprentissage automatique, notamment en apprentissage profond. L'objectif est de modifier l'échelle des données d'entrée pour les rendre homogènes, ce qui améliore et accélère le processus d'apprentissage. Lorsque les données ne sont pas normalisées, les écarts de grandeurs entre les différentes caractéristiques peuvent conduire

à des gradients disproportionnés pendant la descente de gradient, rendant l'optimisation plus difficile et plus longue. Cela peut également provoquer des problèmes de convergence et rendre l'algorithme sensible à l'échelle des caractéristiques.

Pour le jeu de données MNIST, qui contient des images en niveaux de gris, les valeurs des pixels peuvent varier de 0 à 255. La normalisation réduit cette gamme à un intervalle [0, 1], ce qui signifie que les poids du réseau de neurones seront mis à jour de manière plus cohérente. Cela permet d'éviter que certaines caractéristiques dominent d'autres uniquement en raison de leur échelle, permettant ainsi au modèle d'apprendre plus efficacement les patterns.

En Python, cette étape peut être réalisée simplement en utilisant des opérations vectorisées sur les données d'entrée:

```
from sklearn.datasets import fetch_openml

# Chargement du jeu de données MNIST
mnist = fetch_openml('mnist_784', version=1, parser="auto")
X, y = mnist["data"], mnist["target"]

# Normalisation des données en divisant par 255
X_normalized = X / 255.0
```

La normalisation est particulièrement cruciale pour les réseaux de neurones profonds, qui sont sensibles aux échelles des données d'entrée en raison de la complexité de leurs architectures et de la nature chaotique de l'optimisation qu'ils nécessitent.

4.1.2 Fonction d'activation: Softmax

Dans les réseaux de neurones, les fonctions d'activation jouent un rôle crucial. Elles introduisent une non-linéarité essentielle dans le modèle, la fonction Softmax est particulièrement utilisée dans les problèmes de classification multi-classes, comme c'est le cas avec le jeu de données MNIST. Elle transforme les scores (aussi appelés logits) calculés par le réseau en une distribution de probabilités, où chaque probabilité correspond à la chance qu'une entrée appartienne à une classe spécifique. Mathématiquement, elle est exprimée comme suit :

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

où x_i est le score de la classe i et le dénominateur est la somme des exponentielles de tous les scores.

Pour visualiser comment la fonction Softmax agit sur un ensemble de scores, voici un petit exemple en Python :

```python
def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum()

# Exemple d'utilisation
scores = [2.0, 1.0, 0.1]
probabilities = softmax(scores)
print(probabilities)
```

```
[0.65900114 0.24243297 0.09856589]
```

4.1.3 Fonction de perte: Perte d'Entropie Croisée

Nous avons vu précédemment qu'il est nécessaire d'utiliser une fonction de perte afin de quantifier l'erreur de notre réseau de neurones, pour pouvoir la corriger. La sélection d'une fonction de perte appropriée est donc cruciale et dépend fortement du type de problème et de données traités.

Dans le contexte de la classification, notamment avec des données comme MNIST où l'on classe des éléments dans des catégories distinctes, la perte d'entropie croisée (Cross-Entropy Loss) est souvent utilisée. Cette fonction est particulièrement adaptée aux problèmes de classification où les sorties sont des probabilités, elle s'exprime comme suit :

$$L = -\sum_{i=1}^{C} y_i \log(p_i)$$

où :

- C est le nombre de classes.
- y_i est 1 si la classe réelle est i et 0 autrement (one-hot encoding).
- p_i est la probabilité prédite pour la classe i.

```python
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split

def preprocess_data(X, y):
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )
    encoder = OneHotEncoder(sparse_output=False, categories="auto")
    y_train_encoded = encoder.fit_transform(y_train.values.reshape(-1, 1))
    y_test_encoded = encoder.transform(y_test.values.reshape(-1, 1))
    return X_train, X_test, y_train_encoded, y_test_encoded
```

```
X_train, X_test, y_train_encoded, y_test_encoded = preprocess_data(X_normalized, y)

# Calcul de la perte (Cross-Entropy)
def compute_loss(predictions, targets):
    return -np.sum(targets * np.log(predictions)) / predictions.shape[0]
```

4.1.4 Rétropropagation

La rétropropagation dans un réseau de neurones avec une couche d'entrée de 784 neurones (correspondant aux 28x28 pixels d'une image), une couche de sortie de 10 neurones (représentant les 10 classes de chiffres), et utilisant la fonction d'activation Softmax et la perte d'entropie croisée, peut être décrite comme suit :

4.1.4.1 Structure du Réseau

- **Couche d'entrée**: 784 neurones.
- **Couche de sortie**: 10 neurones.
- **Poids (w)**: Matrice de dimensions 10x784.
- **Biais (b)**: Vecteur de 10 éléments.

4.1.4.2 Forward Pass

1. **Calcul des logits**: $Z = Xw + b$ où X est l'entrée du réseau, w est la matrice des poids et b est le vecteur de biais.
2. **Application de la fonction Softmax**: $\hat{Y} = \text{Softmax}(Z)$ où $\hat{Y}$ représente les probabilités prédites.

4.1.4.3 Rétropropagation

1. **Gradient de la fonction de perte par rapport à la sortie**:

$$\frac{\partial L}{\partial \hat{Y}} = \hat{Y} - Y$$

   où Y est le label réel et $\hat{Y}$ la sortie du Softmax.

2. **Gradient par rapport aux logits (Z)**:

$$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial Z} = \hat{Y} - Y$$

3. **Gradient par rapport aux poids (w)**:

$$\frac{\partial L}{\partial w} = \frac{\partial Z}{\partial w} \cdot \frac{\partial L}{\partial Z} = \frac{1}{n} \cdot X^T \cdot (\hat{Y} - Y)$$

4. **Gradient par rapport aux biais (b)**:

$$\frac{\partial L}{\partial b} = \frac{1}{n} \cdot \sum(\hat{Y} - Y)$$

Sachant que l'on calcule la perte sur plusieurs images, il faut diviser par le nombre d'images, n

4.1.4.4 Mise à Jour des Paramètres

Les poids w et les biais b sont mis à jour en soustrayant le produit du taux d'apprentissage (lr) et du gradient correspondant :

- $w = w - \text{lr} \cdot \frac{\partial L}{\partial w}$
- $b = b - \text{lr} \cdot \frac{\partial L}{\partial b}$

4.1.5 Résultats

Nous implémentons alors une boucle d'apprentissage, avec un batch de 32, et une évaluation du modèle

```
# Initialisation des paramètres
def initialize_parameters(input_size, output_size):
    weights = np.random.randn(input_size, output_size) * 0.01
    biases = np.zeros((1, output_size))
    return weights, biases


# Softmax
def softmax(logits):
    exp_logits = np.exp(logits)
    return exp_logits / np.sum(exp_logits, axis=1, keepdims=True)


# Calcul de la perte (Cross-Entropy)
def compute_loss(predictions, targets):
    return -np.sum(targets * np.log(predictions)) / predictions.shape[0]


# Calcul des gradients
def compute_gradient(predictions, targets):
    return predictions - targets
```

```python
# Rétropropagation
def backpropagate(X, gradient, weights, biases, learning_rate):
    dW = np.dot(X.T, gradient) / X.shape[0]
    db = np.sum(gradient, axis=0, keepdims=True) / X.shape[0]
    weights -= learning_rate * dW
    biases -= learning_rate * db
    return weights, biases


# Entraînement du modèle
def train_model(X_train, y_train_encoded, input_size, output_size, learning_rate, epo
    weights, biases = initialize_parameters(input_size, output_size)
    for epoch in range(epochs):
        losses = []

        for i in range(0, X_train.shape[0], batch_size):
            X_train_batch = X_train[i:i + batch_size]
            y_train_batch = y_train_encoded[i:i + batch_size]

            logits = np.dot(X_train_batch, weights) + biases
            predictions = softmax(logits)
            loss = compute_loss(predictions, y_train_batch)
            losses.append(loss)
            gradient = compute_gradient(predictions, y_train_batch)
            weights, biases = backpropagate(X_train_batch, gradient, weights, biases,

        average_loss = np.mean(losses)
        print(f"Epoch {epoch + 1}/{epochs}, Average Loss: {average_loss:.4f}")
    return weights, biases

# Évaluation du modèle
def evaluate_model(X_test, y_test_encoded, weights, biases):
    test_logits = np.dot(X_test, weights) + biases
    test_predictions = softmax(test_logits)
    test_accuracy = np.mean(
        np.argmax(test_predictions, axis=1) == y_test_encoded.argmax(axis=1)
    )
```

```
    print(f"Accuracy on test set: {test_accuracy:.2f}")

input_size = X_train.shape[1]
output_size = y_train_encoded.shape[1]
learning_rate = 0.1
epochs = 20
batch_size = 32

weights, biases = train_model(
    X_train, y_train_encoded, input_size, output_size, learning_rate, epochs, batch_s
)
evaluate_model(X_test, y_test_encoded, weights, biases)
```

```
Epoch 1/20, Average Loss: 0.4159
Epoch 2/20, Average Loss: 0.3130
Epoch 3/20, Average Loss: 0.2955
Epoch 4/20, Average Loss: 0.2859
Epoch 5/20, Average Loss: 0.2796
Epoch 6/20, Average Loss: 0.2749
Epoch 7/20, Average Loss: 0.2713
Epoch 8/20, Average Loss: 0.2683
Epoch 9/20, Average Loss: 0.2658
Epoch 10/20, Average Loss: 0.2637
Epoch 11/20, Average Loss: 0.2618
Epoch 12/20, Average Loss: 0.2602
Epoch 13/20, Average Loss: 0.2587
Epoch 14/20, Average Loss: 0.2573
Epoch 15/20, Average Loss: 0.2561
Epoch 16/20, Average Loss: 0.2550
Epoch 17/20, Average Loss: 0.2540
Epoch 18/20, Average Loss: 0.2530
Epoch 19/20, Average Loss: 0.2521
Epoch 20/20, Average Loss: 0.2513
Accuracy on test set: 0.92
```