Introduction au Machine Learning

Puisque nous apprenons `Python`, il est certain que nous devons également parler d'apprentissage automatique. En effet, de nombreuses bibliothèques dans ce domaine sont écrites en Python. Commençons par les installer et jouons un peu avec elles.

TensorFlow

□□□□□

```
$ pip install tensorflow
ERREUR : Impossible de trouver une version qui satisfait la requête tensorflow
ERREUR : Aucune distribution correspondante trouvée pour tensorflow

$ type python
python est un alias de `/usr/local/Cellar/python@3.9/3.9.1_6/bin/python3'
```

Cependant, `Tensorflow 2` ne supporte que les versions `Python 3.5–3.8`. Nous utilisons la version `3.9`.

```
 % type python3
python3 est /usr/bin/python3
% python3 -V
Python 3.8.2
```

J'ai remarqué que la version de `python3` sur mon système est `3.8.2`. Où est installé le `pip` correspondant à cette version de Python ?

```
% python3 -m pip -V
pip 21.0.1 de /Users/lzw/Library/Python/3.8/lib/python/site-packages/pip (python 3.8)
```

Voici le `pip` correspondant. Je vais donc modifier le fichier `.zprofile`. Récemment, j'ai changé mon `shell`. Le `.zprofile` est l'équivalent de l'ancien `.bash_profile`. J'ajoute une ligne.

```
alias pip3=/Users/lzw/Library/Python/3.8/bin/pip3
```

Ainsi, nous utilisons `python3` et `pip3` pour jouer avec `Tensorflow`.

```
% pip3 install tensorflow
...
Installation réussie : absl-py-0.12.0 astunparse-1.6.3 cachetools-4.2.1 certifi-2020.12.5 chardet-4.0.0
```

J'ai installé de nombreuses bibliothèques. J'ai utilisé un exemple du site officiel.

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])

predictions = model(x_train[:1]).numpy()
print(predictions)
```

Exécutez-le.

```shell
$ /usr/bin/python3 tf.py
Téléchargement des données depuis https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 10s 1us/step
[[ 0.15477428 -0.3877643   0.0994779   0.07474922 -0.26219758 -0.03550266
   0.32226565 -0.37141111  0.10925996 -0.0115255 ]]
```

On peut voir que le jeu de données a été téléchargé, puis les résultats ont été affichés.

Ensuite, examinons un exemple de classification d'images.

```
# TensorFlow et tf.keras
import tensorflow as tf
```

Bibliothèques d’aide

import numpy as np import matplotlib.pyplot as plt

```
print(tf.__version__)
```

Erreur.

```
ModuleNotFoundError: No module named 'matplotlib'
```

Note : Le message d’erreur reste en anglais car il s’agit d’un message technique standard en programmation, souvent utilisé tel quel dans les environnements de développement francophones.

Installez-le.

```
% pip3 install matplotlib
```

C’est correct.

```
$ /usr/bin/python3 image.py
2.4.1
```

Exemple de code pour copier-coller.

```
# TensorFlow et tf.keras
import tensorflow as tf
```

Bibliothèques utilitaires

import numpy as np import matplotlib.pyplot as plt

```
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
print(train_images.shape)
print(len(train_labels))
```

Les résultats ont été affichés. On remarque ici la présence de `train_images`, `train_labels`, `test_images`, et `test_labels`. Cela signifie que les données sont divisées en un ensemble d'entraînement et un ensemble de test.

```
(60000, 28, 28)
60000
```

Ensuite, essayons d'imprimer l'image.

```
print(train_images[0])
```

Voyons les résultats.

```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0  13  73   0
    0   1   4   0   0   0   0   1   1   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   3   0  36 136 127  62
   54   0   0   0   1   3   4   0   0   3]
 [  0   0   0   0   0   0   0   0   0   0   0   0   6   0 102 204 176 134
  144 123  23   0   0   0   0  12  10   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 155 236 207 178
  107 156 161 109  64  23  77 130  72  15]
 [  0   0   0   0   0   0   0   0   0   0   0   1   0  69 207 223 218 216
  216 163 127 121 122 146 141  88 172  66]]
  ....
```

Voici un extrait des résultats.

```
print(len(train_images[0][0]))
```

La sortie est `28`. Il est donc clair qu'il s'agit d'une matrice de largeur 28. Continuons à imprimer.

```
    print(len(train_images[0][0][0])
TypeError: l'objet de type 'numpy.uint8' n'a pas de len()
```

Il est donc très clair. Chaque image est un tableau de dimensions `28*28*3`. La dernière dimension du tableau contient les valeurs RGB. Cependant, nous avons réalisé que notre hypothèse pourrait être erronée.

```
print(train_images[0][1][20])
```

```
0
```

```
print(train_images[0][1])
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Chaque image est représentée par un tableau de 28*28. Après avoir bidouillé un peu, nous avons finalement découvert le secret.

Commençons par examiner le graphique généré.

```
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```

Vous voyez la barre de couleur à droite ? De `0` à `250`. En fait, c'est un dégradé entre deux couleurs. Mais comment sait-elle quelles sont ces deux couleurs ? Où lui avons-nous dit cela ?

Ensuite, nous imprimons également la deuxième image.

```
plt.imshow(train_images[1])
```

C'est très intéressant. Est-ce que c'est par défaut dans la bibliothèque de dépendance de `pyplot` ? Continuons à exécuter le code fourni sur le site officiel.

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
```

```
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

Remarquez que les images ainsi que leurs catégories sont affichées ici. Enfin, nous avons compris le paramètre `cmp`. Si on ne spécifie rien pour `cmp`, cela prendra certainement la couleur que nous avions tout à l'heure. Effectivement.

```
    plt.imshow(train_images[i])
```

Cette fois-ci, nous recherchons `pyplot cmap`. Nous trouvons quelques ressources.

```
    plt.imshow(train_images[i], cmap=plt.cm.PiYG)
```

Voici la version modifiée du code en français :

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)   ## Changement ici
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.Blues)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

Explication des modifications :

- La ligne `plt.subplot(2,5,i+1)` a été modifiée en `plt.subplot(5,5,i+1)` pour afficher les images dans une grille de 5 lignes et 5 colonnes au lieu de 2 lignes et 5 colonnes. Cela permet d'afficher correctement les 25 images dans une disposition plus équilibrée.

Cependant, une erreur s'est produite.

```
ValueError: num doit être compris entre 1 et 10, pas 11
```

Cela signifie quoi exactement. Que signifie vraiment le `5,5,i+1` précédent. Pourquoi cela ne fonctionne plus quand on le change en `2`. Bien que nous ayons intuitivement l'impression que

cela signifie probablement 5 lignes et 5 colonnes. Mais pourquoi cela génère-t-il cette erreur. Comment `11` est-il calculé. Que signifie `num`. Que signifie `10`. On remarque que `2*5=10`. Donc peut-être que l'erreur se produit quand `i=11`. Lorsqu'on change en `for i in range(10):`, on obtient le résultat suivant.

Je jette un coup d'œil rapide à la documentation et je découvre `subplot(nrows, ncols, index, **kwargs)`. Hum, jusqu'ici, c'est assez clair pour nous.

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    # plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.Blues)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

Remarquez que `0 25` est ce qu'on appelle les `xticks`. Lorsque nous zoomons ou dézoomons sur cette boîte, l'affichage change en conséquence.

plot_scale

Remarquez que lors du zoom avant ou arrière, les `xticks` et les `xlabels` peuvent s'afficher différemment.

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

model.fit(train_images, train_labels, epochs=10)

test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)

```
print('\nPrécision du test :', test_acc)
```

Vous avez remarqué ici la manière dont le modèle est défini, en utilisant la classe `Sequential`. Faites attention à ces paramètres : `28,28`, `128`, `relu`, `10`. Notez qu'il est nécessaire de faire `compile` et `fit`. `fit` signifie ajuster ou entraîner le modèle. Notez également que `28,28` correspond à la taille de l'image.

```
Epoch 1/10
1875/1875 [==============================] - 2s 928us/step - loss: 0.6331 - accuracy: 0.7769
Epoch 2/10
1875/1875 [==============================] - 2s 961us/step - loss: 0.3860 - accuracy: 0.8615
Epoch 3/10
1875/1875 [==============================] - 2s 930us/step - loss: 0.3395 - accuracy: 0.8755
Epoch 4/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.3071 - accuracy: 0.8890
Epoch 5/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.2964 - accuracy: 0.8927
Epoch 6/10
1875/1875 [==============================] - 2s 985us/step - loss: 0.2764 - accuracy: 0.8955
Epoch 7/10
1875/1875 [==============================] - 2s 961us/step - loss: 0.2653 - accuracy: 0.8996
Epoch 8/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.2549 - accuracy: 0.9052
Epoch 9/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.2416 - accuracy: 0.9090
Epoch 10/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.2372 - accuracy: 0.9086
313/313 - 0s - loss: 0.3422 - accuracy: 0.8798
```

Précision du test : 0.879800021648407

Le modèle a été entraîné. Modifions les paramètres.

```shell
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(28, activation='relu'),    # 128 -> 28
```

```
    tf.keras.layers.Dense(10)
])
```

Modifiez le premier paramètre de `Dense`.

```
Epoch 1/10
1875/1875 [==============================] - 2s 714us/step - loss: 6.9774 - accuracy: 0.3294
Epoch 2/10
1875/1875 [==============================] - 1s 715us/step - loss: 1.3038 - accuracy: 0.4831
Epoch 3/10
1875/1875 [==============================] - 1s 747us/step - loss: 1.0160 - accuracy: 0.6197
Epoch 4/10
1875/1875 [==============================] - 1s 800us/step - loss: 0.7963 - accuracy: 0.6939
Epoch 5/10
1875/1875 [==============================] - 2s 893us/step - loss: 0.7006 - accuracy: 0.7183
Epoch 6/10
1875/1875 [==============================] - 1s 747us/step - loss: 0.6675 - accuracy: 0.7299
Epoch 7/10
1875/1875 [==============================] - 1s 694us/step - loss: 0.6681 - accuracy: 0.7330
Epoch 8/10
1875/1875 [==============================] - 1s 702us/step - loss: 0.6675 - accuracy: 0.7356
Epoch 9/10
1875/1875 [==============================] - 1s 778us/step - loss: 0.6508 - accuracy: 0.7363
Epoch 10/10
1875/1875 [==============================] - 1s 732us/step - loss: 0.6532 - accuracy: 0.7350
313/313 - 0s - loss: 0.6816 - accuracy: 0.7230
```

Précision du test : 0.7229999899864197

```
On remarque que la `Test accuracy` a changé avant et après. Les `Epoch` sont des logs générés par la fo
```

```
```python
print(train_labels)
[9 0 0 ... 3 0 5]
print(len(train_labels))
60000
```
```

Cela signifie que nous utilisons les chiffres de `0 à 9` pour représenter ces catégories. Par coïncidence, `class_names` en contient également 10.

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

Je vais encore apporter quelques modifications.

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(28, activation='relu'),
    tf.keras.layers.Dense(5)   # 10 -> 5
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=10)
```

Une erreur s'est produite.

```
tensorflow.python.framework.errors_impl.InvalidArgumentError:  Une valeur de label 9 a été reçue, qui es
         [[node sparse_categorical_crossentropy/SparseSoftmaxCrossEntropyWithLogits/SparseSoftmaxCrossEn
```

Pile d'appel de fonction : train_function

```
Il suffit de modifier le troisième paramètre de `Sequential`, en changeant le paramètre de `Dense` à `15

```python
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(28, activation='relu'),
    tf.keras.layers.Dense(15)
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```
```

```
model.fit(train_images, train_labels, epochs=15)  # 10 -> 15

test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)

print('\nPrécision du test :', test_acc)
```

```
Epoch 1/15
1875/1875 [==============================] - 2s 892us/step - loss: 6.5778 - accuracy: 0.3771
Epoch 2/15
1875/1875 [==============================] - 2s 872us/step - loss: 1.3121 - accuracy: 0.4910
Epoch 3/15
1875/1875 [==============================] - 2s 909us/step - loss: 1.0900 - accuracy: 0.5389
Epoch 4/15
1875/1875 [==============================] - 1s 730us/step - loss: 1.0422 - accuracy: 0.5577
Epoch 5/15
1875/1875 [==============================] - 1s 709us/step - loss: 0.9529 - accuracy: 0.5952
Epoch 6/15
1875/1875 [==============================] - 1s 714us/step - loss: 0.9888 - accuracy: 0.5950
Epoch 7/15
1875/1875 [==============================] - 1s 767us/step - loss: 0.8678 - accuracy: 0.6355
Epoch 8/15
1875/1875 [==============================] - 1s 715us/step - loss: 0.8247 - accuracy: 0.6611
Epoch 9/15
1875/1875 [==============================] - 1s 721us/step - loss: 0.8011 - accuracy: 0.6626
Epoch 10/15
1875/1875 [==============================] - 1s 711us/step - loss: 0.8024 - accuracy: 0.6622
Epoch 11/15
1875/1875 [==============================] - 1s 781us/step - loss: 0.7777 - accuracy: 0.6696
Epoch 12/15
1875/1875 [==============================] - 1s 724us/step - loss: 0.7764 - accuracy: 0.6728
Epoch 13/15
1875/1875 [==============================] - 1s 731us/step - loss: 0.7688 - accuracy: 0.6767
Epoch 14/15
1875/1875 [==============================] - 1s 715us/step - loss: 0.7592 - accuracy: 0.6793
Epoch 15/15
1875/1875 [==============================] - 1s 786us/step - loss: 0.7526 - accuracy: 0.6792
313/313 - 0s - loss: 0.8555 - accuracy: 0.6418
```

Précision du test : 0.6417999863624573

Notez que le changement à 15 ne fait pas une grande différence. `tf.keras.layers.Dense(88, activation='

```python
probability_model = tf.keras.Sequential([model,
                                         tf.keras.layers.Softmax()])
```

Voici une prédiction à venir. Notez que `Sequential` est identique à celui mentionné précédemment. Faites également attention aux paramètres `model` et `tf.keras.layers.Softmax()`.

```
probability_model = tf.keras.Sequential([model,
                                         tf.keras.layers.Softmax()])
predictions = probability_model.predict(test_images)

def plot_image(i, predictions_array, true_label, img):
  true_label, img = true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

plt.imshow(img, cmap=plt.cm.binary)

predicted_label = np.argmax(predictions_array)
if predicted_label == true_label:
    color = 'blue'
else:
    color = 'red'

plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)
```

Traduction en français :

```
plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
```

```
                                class_names[true_label]),
                                color=color)
```

Le code reste inchangé car il s'agit d'une chaîne de formatage en Python qui utilise des variables et des fonctions spécifiques. La traduction ne serait pas appropriée ici, car cela pourrait affecter le fonctionnement du code.

```
def plot_value_array(i, predictions_array, true_label):
  true_label = true_label[i]
  plt.grid(False)
  plt.xticks(range(10))
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

thisplot[predicted_label].set_color('rouge')
thisplot[true_label].set_color('bleu')

i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  test_labels)
plt.show()
```

Cela signifie qu'il y a 99 % de chances que cette image soit une `Ankle boot`. Notez que `plot_image` affiche l'image à gauche, tandis que `plot_value_array` génère le graphique à droite.

```
num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions[i], test_labels, test_images)
```

```
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()
```

Notez qu'ici, nous ne faisons qu'afficher davantage de résultats de test. Par conséquent, nous comprenons globalement le processus d'utilisation. Cependant, nous ne savons pas encore comment les calculs sont effectués en arrière-plan. Mais nous savons comment les utiliser. Ils reposent sur le calcul différentiel et intégral. Comment comprendre le calcul différentiel et intégral ?

Par exemple, il y a un nombre entre 1 et 100 que vous devez deviner. À chaque fois que vous proposez un nombre, je vous dis si c'est trop petit ou trop grand. Vous proposez 50. Je dis que c'est trop petit. Vous proposez 80. Je dis que c'est trop grand. Vous proposez 65. Je dis que c'est trop grand. Vous proposez 55. Je dis que c'est trop petit. Vous proposez 58. Je dis, "Oui, c'est le bon nombre."

L'apprentissage automatique consiste à simuler un processus similaire en arrière-plan. Cependant, c'est un peu plus complexe. Il peut s'agir de nombreux `1 à 100`, avec beaucoup de nombres à deviner. En même temps, chaque supposition nécessite de nombreux calculs. Et chaque fois qu'il faut déterminer si c'est trop grand ou trop petit, cela implique également beaucoup de calculs.

PyTorch

Installez-le. Cela prend en charge Python version `3.9`.

```
$ pip install torch torchvision
Collecting torch
  Téléchargement de torch-1.8.0-cp39-none-macosx_10_9_x86_64.whl (120.6 MB)
     |                 | 120.6 MB 224 kB/s
Collecting torchvision
  Téléchargement de torchvision-0.9.0-cp39-cp39-macosx_10_9_x86_64.whl (13.1 MB)
     |                 | 13.1 MB 549 kB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.9/site-packages (from torch) (1.20.1)
Collecting typing-extensions
  Téléchargement de typing_extensions-3.7.4.3-py3-none-any.whl (22 kB)
Requirement already satisfied: pillow>=4.1.1 in /usr/local/lib/python3.9/site-packages (from torchvisio
```

```
Installation des paquets collectés : typing-extensions, torch, torchvision
Installation réussie de torch-1.8.0 torchvision-0.9.0 typing-extensions-3.7.4.3
```

Vérifions cela.

```
import torch
x = torch.rand(5, 3)
print(x)
```

Une erreur s'est produite.

```
Traceback (most recent call last):
  File "torch.py", line 1, in <module>
    import torch
  File "torch.py", line 2, in <module>
    x = torch.rand(5, 3)
AttributeError: le module 'torch' partiellement initialisé n'a pas d'attribut 'rand' (probablement dû à
```

J'ai cherché cette erreur sur Google. Il s'avère que c'était parce que notre fichier s'appelait aussi `torch`. Il y avait un conflit de noms. Après avoir modifié le nom, tout fonctionne correctement.

```
tensor([[0.5520, 0.9446, 0.5543],
        [0.6192, 0.0908, 0.8726],
        [0.0223, 0.7685, 0.9814],
        [0.4019, 0.5406, 0.3861],
        [0.5485, 0.6040, 0.2387]])
```

Trouver un exemple.

```
# -*- coding: utf-8 -*-

import torch
import math
dtype = torch.float
device = torch.device("cpu")
# device = torch.device("cuda:0") # Décommentez cette ligne pour exécuter sur GPU
```

Créer des données d’entrée et de sortie aléatoires

x = torch.linspace(-math.pi, math.pi, 2000, device=device, dtype=dtype) y = torch.sin(x)

Initialisation aléatoire des poids

a = torch.randn((), device=device, dtype=dtype) b = torch.randn((), device=device, dtype=dtype) c = torch.randn((), device=device, dtype=dtype) d = torch.randn((), device=device, dtype=dtype)

```
learning_rate = 1e-6
for t in range(2000):
    # Passe avant : calcul de la prédiction y
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

# Calculer et afficher la perte
loss = (y_pred - y).pow(2).sum().item()
if t % 100 == 99:
    print(t, loss)

# Rétropropagation pour calculer les gradients de a, b, c, d par rapport à la perte
grad_y_pred = 2.0 * (y_pred - y)
grad_a = grad_y_pred.sum()
grad_b = (grad_y_pred * x).sum()
grad_c = (grad_y_pred * x ** 2).sum()
grad_d = (grad_y_pred * x ** 3).sum()

# Mettre à jour les poids en utilisant la descente de gradient
a -= learning_rate * grad_a
b -= learning_rate * grad_b
c -= learning_rate * grad_c
d -= learning_rate * grad_d
```

print(f’Résultat : y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3’)

Exécutez-le.

```shell
99 1273.537353515625
199 849.24853515625
299 567.4786987304688
399 380.30291748046875
499 255.92752075195312
599 173.2559814453125
699 118.2861328125
799 81.72274780273438
899 57.39331817626953
999 41.198158264160156
1099 30.41307830810547
1199 23.227672576904297
1299 18.438262939453125
1399 15.244369506835938
1499 13.113286972045898
1599 11.690631866455078
1699 10.740333557128906
1799 10.105220794677734
1899 9.6804780960083
1999 9.39621353149414
Résultat : y = -0.011828352697193623 + 0.8360244631767273 x + 0.002040589228272438 x^2 + -0.090383656321
```

Voici un exemple de code utilisant uniquement la bibliothèque `numpy` :

```
import numpy as np

# Créer un tableau numpy
array = np.array([1, 2, 3, 4, 5])

# Effectuer des opérations de base
print("Tableau original :", array)
print("Somme des éléments :", np.sum(array))
print("Moyenne des éléments :", np.mean(array))
print("Élément maximum :", np.max(array))
print("Élément minimum :", np.min(array))
```

```
# Multiplier chaque élément par 2
array_multiplied = array * 2
print("Tableau multiplié par 2 :", array_multiplied)

# Créer une matrice 2x2
matrix = np.array([[1, 2], [3, 4]])
print("Matrice :\n", matrix)

# Calculer le déterminant de la matrice
determinant = np.linalg.det(matrix)
print("Déterminant de la matrice :", determinant)
```

Ce code montre quelques opérations de base que vous pouvez effectuer avec `numpy`, comme la création de tableaux, les opérations mathématiques, et le calcul de propriétés matricielles.

```
# -*- coding: utf-8 -*-
import numpy as np
import math
```

Créer des données d'entrée et de sortie aléatoires

x = np.linspace(-math.pi, math.pi, 2000) y = np.sin(x)

Initialisation aléatoire des poids

a = np.random.randn() b = np.random.randn() c = np.random.randn() d = np.random.randn()

```
learning_rate = 1e-6
for t in range(2000):
    # Passe avant : calcul de la prédiction y
    # y = a + b x + c x^2 + d x^3
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

# Calculer et afficher la perte
```

```
loss = np.square(y_pred - y).sum()
if t % 100 == 99:
    print(t, loss)

# Rétropropagation pour calculer les gradients de a, b, c, d par rapport à la perte
grad_y_pred = 2.0 * (y_pred - y)
grad_a = grad_y_pred.sum()
grad_b = (grad_y_pred * x).sum()
grad_c = (grad_y_pred * x ** 2).sum()
grad_d = (grad_y_pred * x ** 3).sum()

# Mise à jour des poids
a -= learning_rate * grad_a
b -= learning_rate * grad_b
c -= learning_rate * grad_c
d -= learning_rate * grad_d

print(f'Résultat : y = {a} + {b} x + {c} x^2 + {d} x^3')
```

Notez qu'il s'agit de deux méthodes de calcul.

Ces deux exemples commencent par générer un ensemble de valeurs pour x et y. Ensuite, on suppose qu'il s'agit d'une équation cubique. Puis, on utilise certaines méthodes pour calculer les coefficients de manière itérative. Comment fonctionnent ces algorithmes ? On remarque qu'il y a une boucle de `2000` itérations, où à chaque étape, l'ajustement devient plus précis. Nous n'entrerons pas dans les détails pour l'instant.

Enfin

Actuellement, nous ne comprenons pas comment les calculs derrière l'apprentissage automatique fonctionnent. Cependant, cela n'est pas crucial pour le moment. Avec les connaissances similaires mentionnées ci-dessus, nous pouvons déjà accomplir beaucoup de choses. Nous pouvons également utiliser l'apprentissage automatique pour traiter du texte, de l'audio, etc. Une fois que nous aurons exploré plusieurs dizaines d'exemples, il sera temps d'apprendre les principes sous-jacents.

Exercices

- Les étudiants explorent de la même manière que ci-dessus.