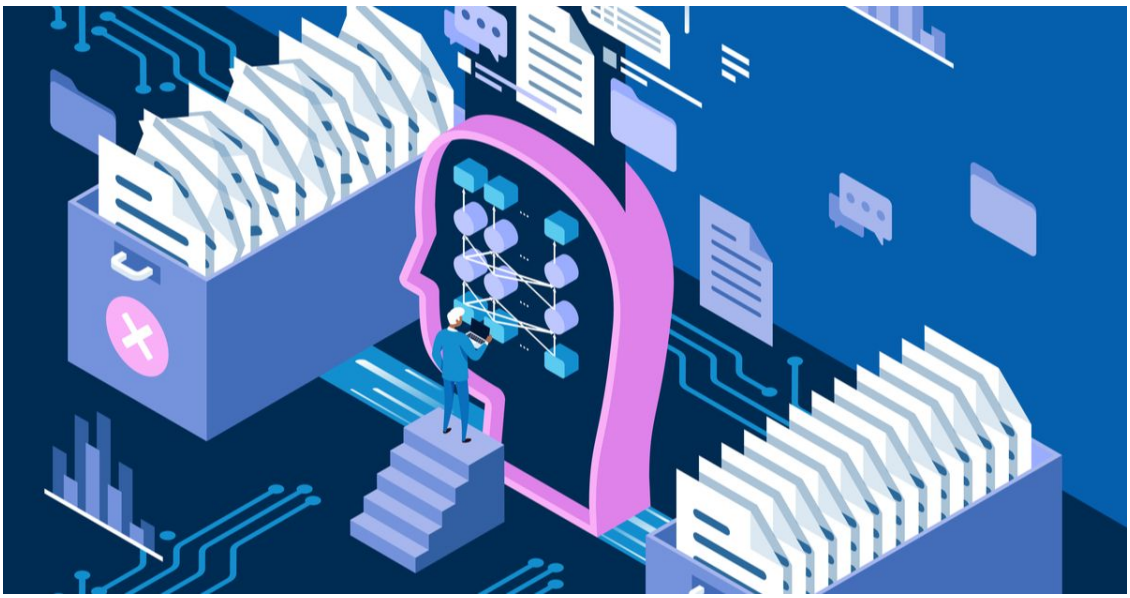
PyTorch _Building_aSimple_NeuralNetw

September 27, 2024

0.1 Tutoriel PyTorch : Construire un réseau neuronal simple à partir de zéro

Découvrez les bases de PyTorch, tout en jetant un coup d'œil sur le fonctionnement des réseaux neuronaux. Commencez à utiliser PyTorch dès aujourd'hui.



Dans ce tutoriel PyTorch, nous allons couvrir les fonctions de base qui alimentent les réseaux neuronaux et construire les nôtres à partir de zéro. L'objectif principal de cet article est de démontrer les bases de PyTorch, une bibliothèque tensorielle optimisée pour l'apprentissage profond, tout en vous fournissant des informations détaillées sur le fonctionnement des réseaux neuronaux.

Note: Check out this DataCamp workspace to follow along with the code written in this article.

0.2 Qu'est-ce qu'un réseau neuronal ?

Les réseaux neuronaux sont également appelés réseaux neuronaux artificiels (RNA). Cette architecture est à la base de l'apprentissage profond, qui n'est qu'un sous-ensemble de l'apprentissage automatique concernant les algorithmes qui s'inspirent de la structure et de la fonction du cerveau humain.En d'autres termes, les réseaux neuronaux constituent la base des architectures qui imitent la façon dont les neurones biologiques se signalent les uns aux autres.

Par conséquent, vous trouverez souvent des ressources qui consacrent les cinq premières minutes à la représentation de la structure neuronale du cerveau humain, afin de vous aider à conceptualiser

le fonctionnement visuel d'un réseau neuronal. Mais si vous ne disposez pas de cinq minutes supplémentaires, il est plus facile de définir un réseau neuronal comme une fonction qui associe des entrées à des sorties souhaitées.

L'architecture générique du réseau neuronal se compose des éléments suivants :

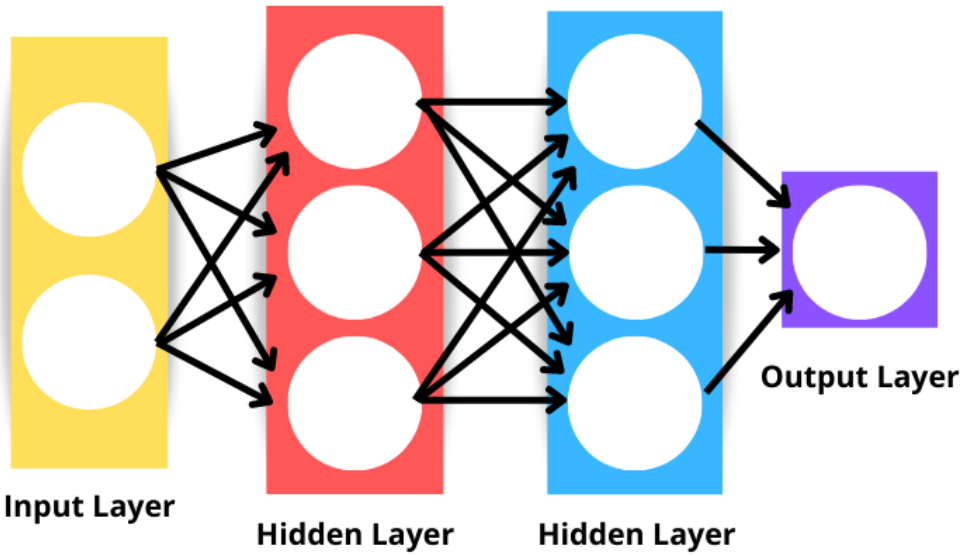
Couche d'entrée : Les données sont introduites dans le réseau par la couche d'entrée. Le nombre de neurones dans la couche d'entrée est équivalent au nombre de caractéristiques dans les données. Techniquement, la couche d'entrée n'est pas considérée comme l'une des couches du réseau, car aucun calcul n'a lieu à ce stade.

Couche cachée : Les couches situées entre les couches d'entrée et de sortie sont appelées couches cachées. Un réseau peut avoir un nombre arbitraire de couches cachées. Plus il y a de couches cachées, plus le réseau est complexe.

Couche de sortie : La couche de sortie est utilisée pour effectuer une prédiction.

Les neurones : Chaque couche comporte un ensemble de neurones qui interagissent avec les neurones des autres couches.

Fonction d'activation : Effectue des transformations non linéaires pour aider le modèle à apprendre des schémas complexes à partir des données.



Notez que le réseau neuronal représenté dans l'image ci-dessus serait considéré comme un réseau neuronal à trois couches et non à quatre - ceci parce que nous n'incluons pas la couche d'entrée comme une couche. Ainsi, le nombre de couches d'un réseau est le nombre de couches cachées plus la couche de sortie.

0.3 Comment fonctionnent les réseaux neuronaux ?

Décomposons l'algorithme en éléments plus petits pour mieux comprendre le fonctionnement des réseaux neuronaux.

0.3.1 Initialisation du poids

L'initialisation des poids est la première composante de l'architecture du réseau neuronal. Les poids initiaux que nous fixons définissent le point de départ du processus d'optimisation du modèle de réseau neuronal.

La manière dont nous définissons nos poids est importante, en particulier lorsque nous construisons des réseaux profonds. En effet, les réseaux profonds sont plus susceptibles de souffrir du problème d'explosion ou de disparition du gradient. Les problèmes de disparition et d'explosion du gradient sont deux concepts qui dépassent le cadre de cet article, mais ils décrivent tous deux un scénario dans lequel l'algorithme ne parvient pas à apprendre.

Bien que l'initialisation des poids ne résolve pas complètement le problème de l'évanouissement ou de l'explosion du gradient, elle contribue certainement à le prévenir.

Voici quelques approches courantes de l'initialisation des poids :

0.3.2 Initialisation du zéro

L'initialisation zéro signifie que les poids sont initialisés à zéro. Ce n'est pas une bonne solution car notre réseau neuronal ne parviendrait pas à briser la symétrie - il n'apprendrait pas.

Lorsqu'une valeur constante est utilisée pour initialiser les poids d'un réseau neuronal, on peut s'attendre à ce que ses performances soient médiocres, car toutes les couches apprendront la même chose. Si toutes les sorties des unités cachées ont la même influence sur le coût, les gradients seront identiques.

0.3.3 Initialisation aléatoire

L'initialisation aléatoire brise la symétrie, ce qui signifie qu'elle est meilleure que l'initialisation zéro, mais certains facteurs peuvent dicter la qualité globale du modèle.

Par exemple, si les poids sont initialisés de manière aléatoire avec des valeurs élevées, on peut s'attendre à ce que chaque multiplication de la matrice donne une valeur nettement plus élevée. Lorsqu'une fonction d'activation sigmoïde est appliquée dans de tels scénarios, le résultat est une valeur proche de un, ce qui ralentit le taux d'apprentissage.

Un autre scénario dans lequel l'initialisation aléatoire peut poser des problèmes est celui où les poids sont initialisés aléatoirement à de petites valeurs. Dans ce cas, chaque multiplication de matrice produira des valeurs significativement plus petites, et l'application d'une fonction sigmoïde produira une valeur plus proche de zéro, ce qui ralentit également le taux d'apprentissage.

0.3.4 Initialisation de Xavier/Glorot

L'initialisation Xavier ou Glorot - quel que soit son nom - est une approche heuristique utilisée pour initialiser les poids. Il est courant de voir cette approche d'initialisation lorsqu'une fonction d'activation tanh ou sigmoïde est appliquée à la moyenne pondérée. Cette approche a été proposée pour la première fois en 2010 dans un document de recherche intitulé Understanding the difficulty

of training deep feedforward neural networks par Xavier Glorot et Yoshua Bengio. Cette technique d'initialisation vise à maintenir la variance du réseau égale afin d'éviter que les gradients n'explosent ou ne disparaissent.

0.3.5 Initialisation du He/Kaiming

L'initialisation He ou Kaiming est une autre approche heuristique. La différence avec l'heuristique He et Xavier est que l'initialisation He utilise un facteur d'échelle différent pour les poids qui prend en compte la non-linéarité des fonctions d'activation.Ainsi, lorsque la fonction d'activation ReLU est utilisée dans les couches, l'initialisation He est l'approche recommandée. Vous pouvez en savoir plus sur cette approche dans Delving Deep into Rectifiers : Surpassing Human-Level Performance on ImageNet Classification par He et al.

0.3.6 Propagation vers l'avant

Les réseaux neuronaux fonctionnent en prenant une moyenne pondérée plus un terme de biais et en appliquant une fonction d'activation pour ajouter une transformation non linéaire. Dans la formule de la moyenne pondérée, chaque poids détermine l'importance de chaque caractéristique (c'est-à-dire sa contribution à la prédiction de la sortie).

$$z = (X_1 \cdot W_1 + X_2 \cdot W_2 + \ldots + X_n \cdot W_n) + b$$

La formule ci-dessus est la moyenne pondérée plus un terme de biais où, - z est la somme pondérée de l'entrée d'un neurone - Wn représente les poids - Xn représente les variables indépendantes,et - b est le terme de biais.

Si la formule vous semble familière, c'est parce qu'il s'agit d'une régression linéaire. Sans l'introduction de la non-linéarité dans les neurones, nous aurions une régression linéaire, qui est un modèle simple. La transformation non linéaire permet à notre réseau neuronal d'apprendre des modèles complexes.

0.4 Fonctions d'activation

Nous avons déjà fait allusion à certaines fonctions d'activation dans la section sur l'initialisation des poids, mais vous connaissez maintenant leur importance dans l'architecture d'un réseau neuronal.

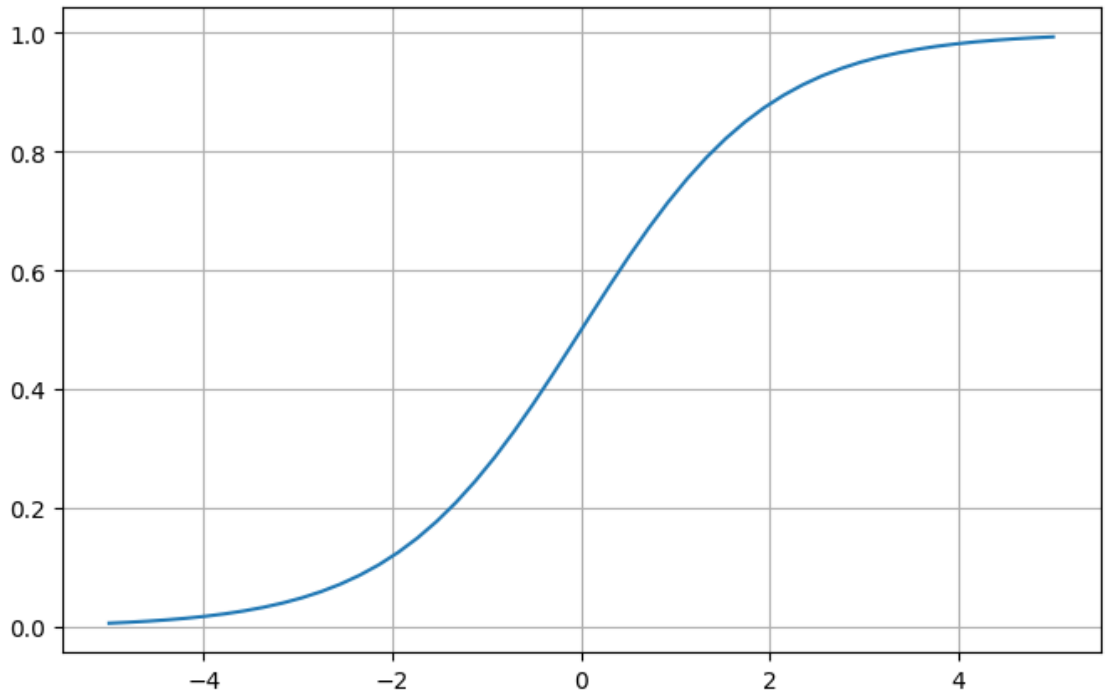
Approfondissons certaines fonctions d'activation courantes que vous êtes susceptible de voir lorsque vous lisez des articles de recherche et le code d'autres personnes.

Sigmoid

La fonction sigmoïde se caractérise par une courbe en forme de "S" limitée entre les valeurs zéro et un. Il s'agit d'une fonction différentiable, ce qui signifie que la pente de la courbe peut être trouvée en deux points quelconques, et monotone, ce qui signifie qu'elle n'est ni entièrement croissante ni entièrement décroissante. La fonction d'activation sigmoïde est généralement utilisée pour les problèmes de classification binaire.

La fonction sigmoïde se caractérise par une courbe en forme de “S” limitée entre les valeurs zéro et un. Il s’agit d’une fonction différentiable, ce qui signifie que la pente de la courbe peut être trouvée en deux points quelconques, et monotone, ce qui signifie qu’elle n’est ni entièrement croissante ni entièrement décroissante. La fonction d’activation sigmoïde est généralement utilisée pour les problèmes de classification binaire.

[1]:
```
# Sigmoid function in Python
import matplotlib.pyplot as plt
import numpy as np




x = np.linspace(-5, 5, 50)
z = 1/(1 + np.exp(-x))




plt.subplots(figsize=(8, 5))
plt.plot(x, z)
plt.grid()
plt.show()
```



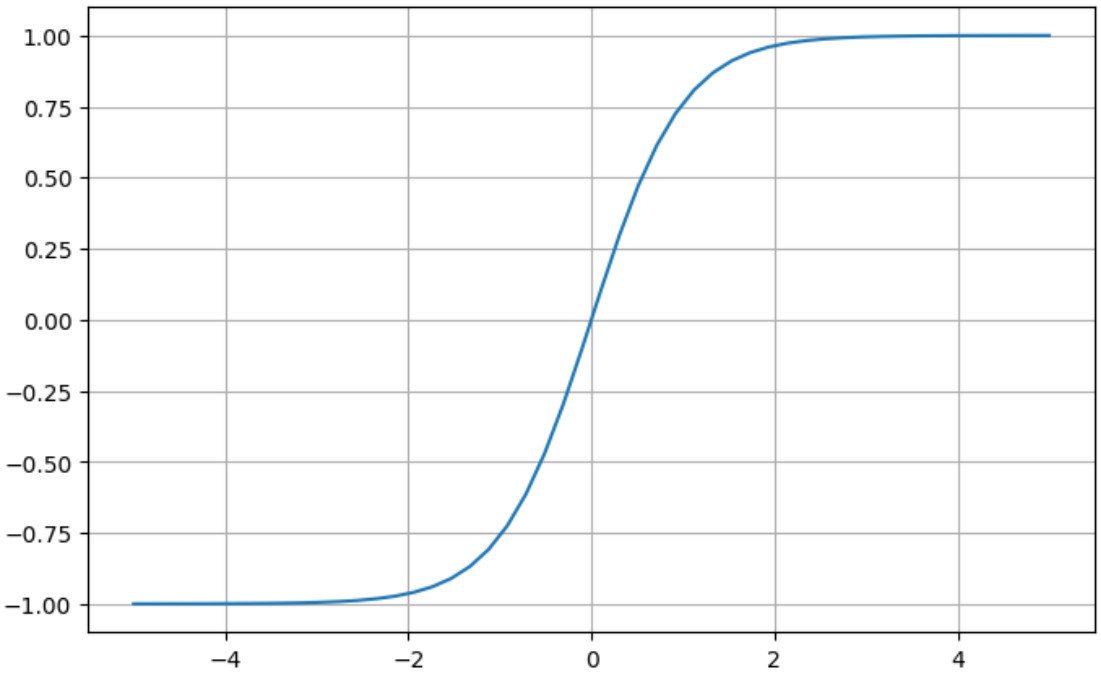
0.4.1 Tanh

La tangente hyperbolique (tanh) a la même courbe en forme de “S” que la fonction sigmoïde, sauf que les valeurs sont limitées entre -1 et 1. Ainsi, les petites entrées sont mappées plus près de -1, et les entrées plus grandes sont mappées plus près de 1.

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

Voici un exemple de fonction tanh visualisée à l’aide de Python :

[2]:
```
# tanh function in Python
import matplotlib.pyplot as plt
import numpy as np




x = np.linspace(-5, 5, 50)
z = np.tanh(x)


plt.subplots(figsize=(8, 5))
plt.plot(x, z)
plt.grid()
plt.show()
```



0.4.2 Softmax

La fonction softmax est généralement utilisée comme fonction d'activation dans la couche de sortie. Il s'agit d'une généralisation de la fonction sigmoïde à plusieurs dimensions. Elle est donc utilisée dans les réseaux neuronaux pour prédire l'appartenance à une classe sur plus de deux étiquettes.

0.4.3 Rectified Linear Unit (ReLU)

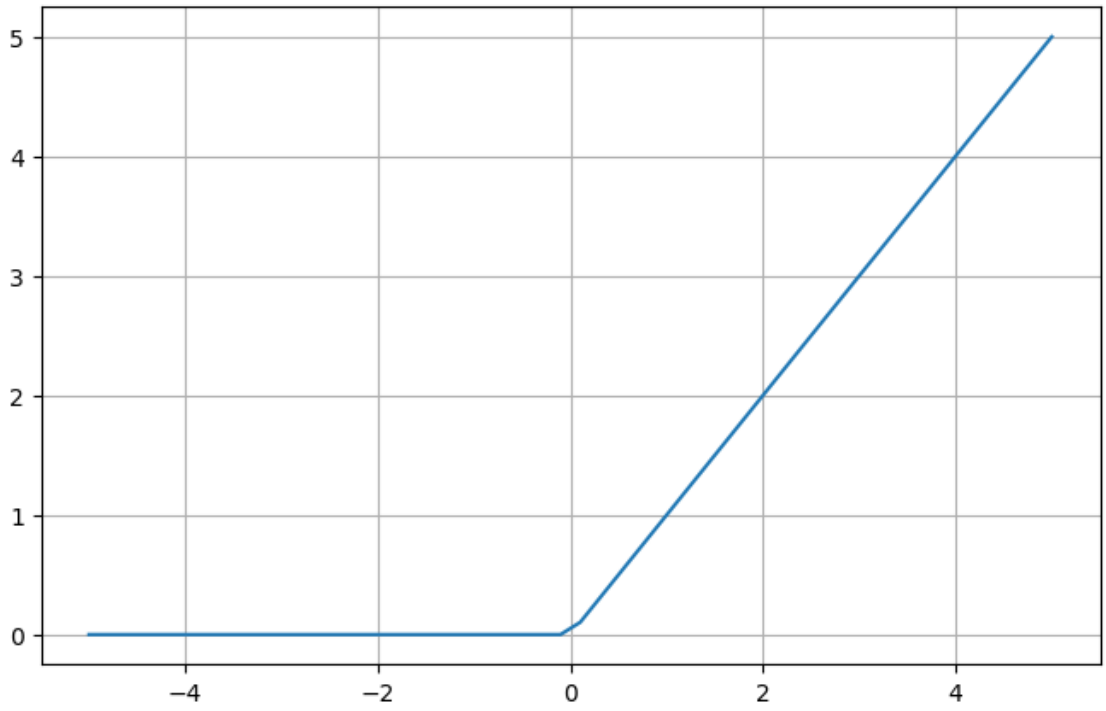
L'utilisation de la fonction sigmoïde ou tanh pour construire des réseaux neuronaux profonds est risquée car ils sont plus susceptibles de souffrir du problème du gradient de disparition. La fonction d'activation de l'unité linéaire rectifiée (ReLU) est apparue comme une solution à ce problème et est souvent la fonction d'activation par défaut de plusieurs réseaux neuronaux.

Voici un exemple visuel de la fonction ReLU en Python :

[3]:
```
# ReLU in Python
import matplotlib.pyplot as plt
import numpy as np


x = np.linspace(-5, 5, 50)
z = [max(0, i) for i in x]


plt.subplots(figsize=(8, 5))
plt.plot(x, z)
```

```
plt.grid()
plt.show()
```

```
The history saving thread hit an unexpected error (OperationalError('attempt to
write a readonly database')).History will not be written to the database.
```



La fonction ReLU est limitée entre zéro et l'infini : notez que pour les valeurs d'entrée inférieures ou égales à zéro, la fonction renvoie zéro, et pour les valeurs supérieures à zéro, la fonction renvoie la valeur d'entrée fournie (c'est-à-dire que si vous entrez deux, la fonction renvoie deux). En fin de compte, la fonction ReLU se comporte de manière extrêmement similaire à une fonction linéaire, ce qui la rend beaucoup plus facile à optimiser et à mettre en œuvre.

Le processus qui va de la couche d'entrée à la couche de sortie est connu sous le nom de "forward pass" ou "forward propagation". Au cours de cette phase, les sorties générées par le modèle sont utilisées pour calculer une fonction de coût afin de déterminer les performances du réseau neuronal après chaque itération. Ces informations sont ensuite transmises au modèle pour corriger les poids afin que le modèle puisse faire de meilleures prédictions dans le cadre d'un processus connu sous le nom de rétropropagation.

0.5 Backpropagation

À la fin de la première passe avant, le réseau fait des prédictions en utilisant les poids initialisés, qui ne sont pas ajustés. Il est donc très probable que les prédictions faites par le modèle ne soient pas exactes. En utilisant la perte calculée à partir de la propagation vers l'avant, nous transmettons des informations au réseau pour affiner les poids dans le cadre d'un processus connu sous le nom

de rétropropagation. En fin de compte, nous utilisons la fonction d'optimisation pour nous aider à identifier les poids susceptibles de réduire le taux d'erreur, ce qui rend le modèle plus fiable et augmente sa capacité à se généraliser à de nouvelles instances. Les mathématiques de ce fonctionnement dépassent le cadre de cet article, mais le lecteur intéressé pourra en apprendre davantage sur la rétropropagation dans notre cours Introduction à l'apprentissage profond en Python.

0.5.1 Tutoriel PyTorch : Une démonstration pas à pas de la construction d'un réseau neuronal à partir de zéro

Dans cette section de l'article, nous allons construire un modèle simple de réseau neuronal artificiel à l'aide de la bibliothèque PyTorch. Consultez cet espace de travail DataCamp pour suivre le code PyTorch est l'une des bibliothèques les plus populaires pour l'apprentissage profond. Elle offre une expérience de débogage beaucoup plus directe que TensorFlow. Il présente plusieurs autres avantages tels que la formation distribuée, un écosystème robuste, la prise en charge du cloud, la possibilité d'écrire du code prêt à la production, etc. Vous pouvez en savoir plus sur PyTorch dans le parcours de compétences Introduction à l'apprentissage profond avec PyTorch.

0.5.2 Définition et préparation des données

The dataset we will be using in our tutorial is make_circles from scikit-learn - see the documentation. It's a toy dataset containing a large circle with a smaller circle in a two-dimensional plane and two features. For our demonstration, we used 10,000 samples and added a 0.05 standard deviation of Gaussian noise to the data.

Avant de construire notre réseau neuronal, il est conseillé de diviser nos données en ensembles de formation et de test afin d'évaluer les performances du modèle sur des données inédites.

[4]:
```
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles
from sklearn.model_selection import train_test_split

# Create a dataset with 10,000 samples.
X, y = make_circles(n_samples = 10000,
                    noise= 0.05,
                    random_state=26)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33,␣
 ↪random_state=26)

# Visualize the data.
fig, (train_ax, test_ax) = plt.subplots(ncols=2, sharex=True, sharey=True,␣
 ↪figsize=(10, 5))
train_ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.Spectral)
train_ax.set_title("Training Data")
train_ax.set_xlabel("Feature #0")
train_ax.set_ylabel("Feature #1")

test_ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test)
test_ax.set_xlabel("Feature #0")
```

```
test_ax.set_title("Testing data")
plt.show()
```

L'étape suivante consiste à convertir les données d'entraînement et de test des tableaux NumPy en tenseurs PyTorch. Pour ce faire, nous allons créer un jeu de données personnalisé pour nos fichiers d'entraînement et de test. Nous allons également utiliser le module Dataloader de PyTorch afin de pouvoir entraîner nos données par lots. Voici le code :

[20]:
```
import warnings
warnings.filterwarnings("ignore")

!pip install torch -q

import torch
import numpy as np
from torch.utils.data import Dataset, DataLoader

# Convert data to torch tensors
class Data(Dataset):
    def __init__(self, X, y):
        self.X = torch.from_numpy(X.astype(np.float32))
        self.y = torch.from_numpy(y.astype(np.float32))
        self.len = self.X.shape[0]

    def __getitem__(self, index):
        return self.X[index], self.y[index]
```

```
    def __len__(self):
        return self.len

batch_size = 64

# Instantiate training and test data
train_data = Data(X_train, y_train)
train_dataloader = DataLoader(dataset=train_data, batch_size=batch_size,␣
 ↪shuffle=True)

test_data = Data(X_test, y_test)
test_dataloader = DataLoader(dataset=test_data, batch_size=batch_size,␣
 ↪shuffle=True)

# Check it's working
for batch, (X, y) in enumerate(train_dataloader):
    print(f"Batch: {batch+1}")
    print(f"X shape: {X.shape}")
    print(f"y shape: {y.shape}")
    break


"""
Batch: 1
X shape: torch.Size([64, 2])
y shape: torch.Size([64])
"""
```

```
Batch: 1
X shape: torch.Size([64, 2])
y shape: torch.Size([64])
```

[20]: `'\nBatch: 1\nX shape: torch.Size([64, 2])\ny shape: torch.Size([64])\n'`

Ce code importe les bibliothèques nécessaires telles que warnings, torch, numpy et DataLoader. Il définit une classe appelée Data qui convertit les données d'entrée en tenseurs torch. La classe possède trois méthodes : init qui initialise la classe avec les données d'entrée, getitem qui renvoie les données à un index spécifique et len qui renvoie la longueur des données. Le code instancie ensuite les données d'entraînement et de test à l'aide de la classe Data et du DataLoader. Le DataLoader est utilisé pour charger les données par lots de taille 64 et pour mélanger les données. Enfin, le code vérifie que le DataLoader fonctionne en imprimant la forme du premier lot de données. La bibliothèque warnings est utilisée pour ignorer tout avertissement pouvant survenir pendant l'exécution du code. La commande !pip install torch -q installe la bibliothèque torch sans afficher de résultat. La bibliothèque numpy est utilisée pour convertir les données d'entrée en tableaux numpy avant de les convertir en tenseurs torch. Globalement, ce code convertit les données d'entrée en tenseurs torch et les charge en lots à l'aide de DataLoader pour un entraînement et un test efficaces des modèles d'apprentissage automatique.

0.6 Mise en œuvre de réseaux neuronaux et formation de modèles

Nous allons mettre en œuvre un réseau neuronal simple à deux couches qui utilise la fonction d'activation ReLU (torch.nn.functional.relu). Pour ce faire, nous allons créer une classe appelée NeuralNetwork qui hérite de nn.Module, la classe de base de tous les modules de réseaux neuronaux construits dans PyTorch.

[27]:
```
import torch
from torch import nn
from torch import optim

input_dim = 2
hidden_dim = 10
output_dim = 1

class NeuralNetwork(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(NeuralNetwork, self).__init__()
        self.layer_1 = nn.Linear(input_dim, hidden_dim)
        nn.init.kaiming_uniform_(self.layer_1.weight, nonlinearity="relu")
        self.layer_2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = torch.nn.functional.relu(self.layer_1(x))
        x = torch.nn.functional.sigmoid(self.layer_2(x))

        return x

model = NeuralNetwork(input_dim, hidden_dim, output_dim)
print(model)


"""
NeuralNetwork(
  (layer_1): Linear(in_features=2, out_features=10, bias=True)
  (layer_2): Linear(in_features=10, out_features=1, bias=True)
)
"""
```

```
NeuralNetwork(
  (layer_1): Linear(in_features=2, out_features=10, bias=True)
  (layer_2): Linear(in_features=10, out_features=1, bias=True)
)
```

[27]:
```
'\nNeuralNetwork(\n  (layer_1): Linear(in_features=2, out_features=10,
bias=True)\n  (layer_2): Linear(in_features=10, out_features=1, bias=True)\n)\n'
```

Ce code définit un modèle de réseau neuronal à l'aide de la bibliothèque PyTorch. Il importe d'abord les bibliothèques nécessaires : torch, nn (abréviation de neural network) et optim (abrévia-

tion de optimizer). Il définit ensuite les dimensions d'entrée, cachée et de sortie du réseau neuronal. Il définit ensuite une classe appelée NeuralNetwork, qui hérite de la classe nn.Module. Cette classe possède une méthode **init** qui initialise les couches du réseau neuronal. La première couche est une couche linéaire (nn.Linear) qui prend en charge la dimension d'entrée et produit des sorties vers la dimension cachée. Les poids de cette couche sont initialisés à l'aide de la méthode d'initialisation uniforme de Kaiming avec une non-linéarité ReLU. La deuxième couche est une autre couche linéaire qui prend en charge la dimension cachée et produit des sorties vers la dimension de sortie. La méthode forward de la classe NeuralNetwork définit le passage vers l'avant du réseau neuronal. Elle prend en charge un tenseur d'entrée x et le fait passer par la première couche avec une fonction d'activation ReLU (torch.nn.functional.relu). La sortie de cette couche passe ensuite par la deuxième couche avec une fonction d'activation sigmoïde (torch.nn.functional.sigmoid). La sortie finale du réseau neuronal est renvoyée. Enfin, une instance de la classe NeuralNetwork est créée avec les dimensions d'entrée, cachée et de sortie spécifiées, et le modèle est imprimé pour montrer la structure du réseau neuronal.

Pour entraîner le modèle, nous devons définir une fonction de perte à utiliser pour calculer les gradients et un optimiseur pour mettre à jour les paramètres. Pour notre démonstration, nous allons utiliser l'entropie croisée binaire et la descente de gradient stochastique avec un taux d'apprentissage de 0,1.

[28]:
```
import torch
import torch.nn as nn
import torch.optim as optim

# Assuming you have a defined model


# Set the learning rate
learning_rate = 0.1

# Define the loss function
loss_fn = nn.BCELoss()

# Define the optimizer, using stochastic gradient descent
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

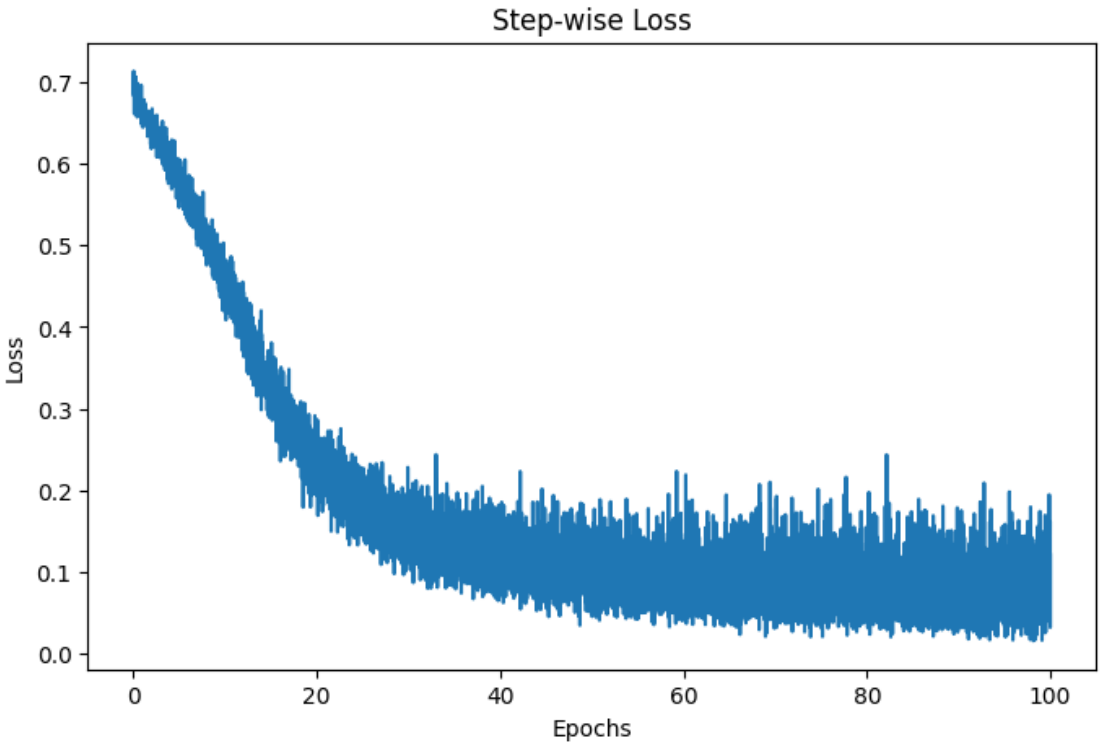
[29]:
```
num_epochs = 100
loss_values = []


for epoch in range(num_epochs):
    for X, y in train_dataloader:
        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
```

```
        pred = model(X)
        loss = loss_fn(pred, y.unsqueeze(-1))
        loss_values.append(loss.item())
        loss.backward()
        optimizer.step()

print("Training Complete")

"""
Training Complete
"""
```

```
Training Complete
```

[29]:
```
'\nTraining Complete\n'
```

[30]:
```
step = np.linspace(0, 100, 10500)

fig, ax = plt.subplots(figsize=(8,5))
plt.plot(step, np.array(loss_values))
plt.title("Step-wise Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
```



La visualisation ci-dessus montre la perte de notre modèle sur 100 époques. Initialement, la perte commence à 0,7 et diminue progressivement - cela nous informe que notre modèle a amélioré ses prédictions au fil du temps. Cependant, le modèle semble plafonner autour de la marque de 60 époques, ce qui peut être dû à une variété de raisons, telles que le modèle peut être dans la région d'un minimum local ou global de la fonction de perte.Néanmoins, le modèle a été entraîné et est prêt à faire des prédictions sur de nouvelles instances - nous verrons comment le faire dans la section suivante.

0.7 Predictions & model evaluation

Faire des prédictions avec notre réseau neuronal PyTorch est assez simple.

[35]:
```
import itertools
import numpy as np
import torch

# Initialize empty lists to store predictions and true labels
y_pred = []
y_test = []

# Initialize variables to track the number of correct predictions and total␣
 ↪predictions
total = 0
correct = 0

# Assuming test_dataloader is defined and model is in evaluation mode
with torch.no_grad():  # No need to compute gradients
    for X, y in test_dataloader:
        outputs = model(X)

        # Convert outputs to binary predictions
        predicted = np.where(outputs.numpy() < 0.5, 0, 1)  # Convert to numpy␣
 ↪array

        # Flatten the predicted values
        predicted = list(itertools.chain(*predicted))

        # Append predictions and true labels
        y_pred.append(predicted)
        y_test.append(y.numpy())

        # Update the total and correct counts
        total += y.size(0)
        correct += (np.array(predicted) == y.numpy()).sum()

# Print accuracy
```

```
print(f'Accuracy of the network on the 3300 test instances: {100 * correct //␣
 ↪total}%')
```

```
Accuracy of the network on the 3300 test instances: 97%
```

[37]:
```
"""
We're not training so we don't need to calculate the gradients for our outputs
"""
with torch.no_grad():
    for X, y in test_dataloader:
        outputs = model(X)
        predicted = np.where(outputs < 0.5, 0, 1)
        predicted = list(itertools.chain(*predicted))
        y_pred.append(predicted)
        y_test.append(y)
        total += y.size(0)
        correct += (predicted == y.numpy()).sum().item()

print(f'Accuracy of the network on the 3300 test instances: {100 * correct //␣
 ↪total}%')
```

```
Accuracy of the network on the 3300 test instances: 97%
```

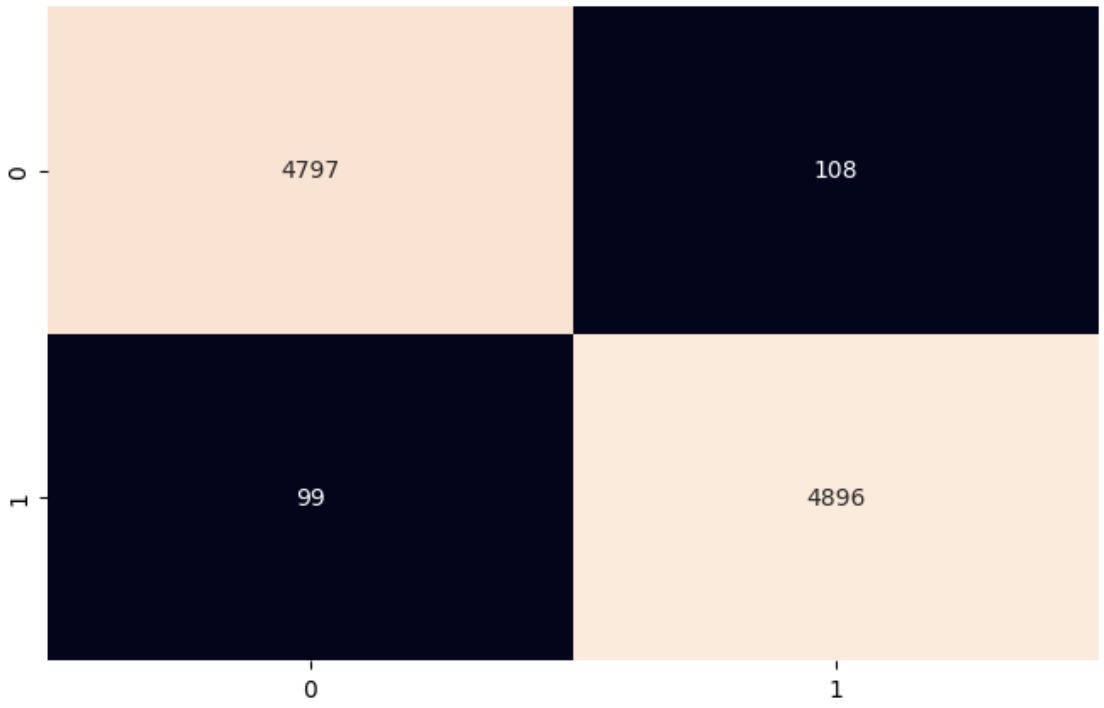
Note : Chaque exécution du code produira un résultat différent, il se peut donc que vous n'obteniez pas les mêmes résultats.

Le code ci-dessus parcourt en boucle les lots de tests, qui sont stockés dans la variable test_dataloader, sans calculer les gradients. Nous prédisons ensuite les instances du lot et stockons les résultats dans une variable appelée outputs. Ensuite, nous déterminons que toutes les valeurs inférieures à 0,5 valent 0 et celles qui sont égales ou supérieures à 0,5 valent 1. Ces valeurs sont ensuite ajoutées à une liste pour nos prédictions. Après cela, nous ajoutons les prédictions réelles des instances du lot à une variable appelée total. Nous calculons ensuite le nombre de prédictions correctes en identifiant le nombre de prédictions correspondant aux classes réelles et en les totalisant. Le nombre total de prédictions correctes pour chaque lot est incrémenté et stocké dans notre variable correct. Pour calculer la précision du modèle global, nous multiplions le nombre de prédictions correctes par 100 (pour obtenir un pourcentage) et le divisons ensuite par le nombre d'instances dans notre ensemble de test. La précision de notre modèle est de 97 %. Nous approfondissons la question en utilisant la matrice de confusion et le rapport classification_report de scikit-learn pour mieux comprendre les performances de notre modèle.

[38]:
```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

import seaborn as sns
```

```
y_pred = list(itertools.chain(*y_pred))
y_test = list(itertools.chain(*y_test))


print(classification_report(y_test, y_pred))




cf_matrix = confusion_matrix(y_test, y_pred)

plt.subplots(figsize=(8, 5))

sns.heatmap(cf_matrix, annot=True, cbar=False, fmt="g")

plt.show()
```

```
              precision    recall  f1-score   support

         0.0       0.98      0.98      0.98      4905
         1.0       0.98      0.98      0.98      4995

    accuracy                           0.98      9900
   macro avg       0.98      0.98      0.98      9900
weighted avg       0.98      0.98      0.98      9900
```



Notre modèle fonctionne plutôt bien. Je vous encourage à explorer le code et à y apporter quelques modifications pour que ce que nous avons abordé dans cet article soit efficace.Dans ce tutoriel PyTorch, nous avons abordé les bases fondamentales des réseaux neuronaux et utilisé PyTorch, une bibliothèque Python pour l'apprentissage profond, pour mettre en œuvre notre réseau.

Nous avons utilisé l'ensemble de données circle's dataset de scikit-learn pour entraîner un réseau neuronal à deux couches pour la classification. Nous avons ensuite effectué des prédictions sur les données et évalué nos résultats à l'aide de la métrique de précision.

[ ]: