coerchon_habasque_dlts_tp3

26 Novembre 2024

[1]:
```
import numpy as np
import h5py
import matplotlib.pyplot as plt
import random
import seaborn as sns

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader

from sklearn.metrics import confusion_matrix, classification_report

# Juste pour débugger mon kernel python
import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'
```

1 TP3: Reconnaissance de signaux de communication par apprentissage profond

Listez les noms des étudiants (2 au maximum) ayant participé à ce notebook dans la cellule suivante (prénom, nom). Au moment du rendu, le notebook doit être nommé nom1_nom2_dlts_tp3.ipynb

2 séances de TP sur ce sujet : le 6 novembre (1h00), le 13 novembre (3h). Le cours du 19 novembre sera partagé en 3 : cours sur la séparation de sources audio / présentation des mini projets et organisation des soutenances / fin de ce TP. Deadline pour le rendu du TP: 26 novembre 2024, 13h59, par mail à deepetsignal.mva@gmail.com

Pour installer les paquets nécessaires à la réalisation de ce TP vous pouvez utiliser dans le notebook

`!pip install \< nom_du_paquet \>`

merci de regrouper toutes les installations dans la première cellule du notebook. Essayez de faire en sorte que votre notebook puisse se lire comme un compte rendu, évitez de laisser du code mort et prennez le temps de commenter vos observations et résultats.

1.1 Problématique

On cherche à identifier un type d'émetteur de communication à partir de l'observation d'un signal provenant de l'émetteur de 2048 échantillons IQ (In Phase / Quadrature) ie le signal prend des valeurs complexes. On représente la partie réelle et la partie imaginaire du signal par deux canaux réel d'un signal multivarié.

L'émetteur peut provenir de 6 catégories différentes. Les paramètres différenciant les différentes catégories sont - le type de modulation - la présence ou non de séquences pilotes et le cas échéant la structure de trame pilotes / données - le débit de la transmission

Les signaux se propagent en champs libre et sont enregistrés par une antenne. Le signal reçu est transposé en bande de base c'est à dire que si le signal est transmis autour d'une fréquence centrale f0, une première étape de traitement du signal à la réception recentre le signal autour de la fréquence 0.

Les différents signaux observés dans ce TP sont entachés de différentes erreurs caractéristiques de la propagation électromagnétique comme : - modification aléatoire de la phase du signal lors de la transmission - imperfection de la transposition en bande de base qui laisse le signal transposé à une fréquence df0 « f0 - présence d'interférence entre les symboles transmis (dûes par exemple à plusieurs chemins de propagation) - présence d'un bruit blanc additif gaussien

Le niveau du bruit relativement à celui du signal utile est décrit par le SNR (Signal to Noise Ratio) et exprimé en dB. On suppose que le SNR est connu lors de l'acquisition d'un signal. Lors de ce TP nous rencontrerons 4 niveaux de SNR: 30 dB (facile), 20 dB, 10 dB et 0 dB (en espérant qu'on puisse faire quelque chose de ces données). Un de nos objectifs sera de qualifier la performance des algorithmes mis en place en fonction du SNR.

Les objectifs de ce TP sont:

1. Définir une ou plusieurs architectures de réseaux de neurones profonds et les implémenter en PyTorch
2. Entrainer ces architectures, la fonction de perte employée pourra être la log vraisemblance négative: https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html.
3. Qualifier les performances de votre réseau de neurones sur l'ensemble de test via:
   - Le calcul de l'accuracy implémentée par exemple dans le package TorchMetrics (https://torchmetrics.readthedocs.io/en/stable/classification/accuracy.html)
   - La réalisation d'un graphique accuracy vs SNR
   - La réalisation des matrices de confusion entre les différentes classes pour les différents SNR (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html#sklearn.metrics.confusion_
4. Rapporter les performances de vos architectures à leur complexité, la complexité d'un réseau de neurones étant ici résumé au nombre de paramètres libres qu'il fait intervenir

Durant l'entraînement on observera l'évolution de la fonction de perte et de l'accuracy sur l'ensemble d'entraînement et sur l'ensemble de validation.

Les 4 premières parties sont un échauffement sur lequel vous pouvez passer vite si vous êtes à l'aise avec le sujet. Le gros du travail est dans la partie 5 "Entraînemenent d'un réseau de neurones".

Surtout privilégiez dans un premier temps la simplicité quitte à complexifier votre approche ensuite pour doper ses performances. Ne restez pas bloqué sur des réseaux qui mettent "trop de temps à

apprendre”

1.2 Chargement des données en numpy

Le TP est composé de trois jeux de données : - train.hdf5 destiné à nourrir l'entrainement de réseaux de neurones - test.hdf5 destiné à évaluer les algorithmes après entrainement - samples.hdf5 qui est beaucoup plus petit que train.hdf5 et destiné à servir de modèle de données dans une phase de prototypage des algorithmes et de la pipeline d'entrainement

Les trois jeux de données sont au format hdf5 qui peut être manipulé via l'API Python h5py https://docs.h5py.org/en/stable/quick.html. Un fichier hdf5 est consitué d'une arborescence de datasets et de groups. Un dataset hdf5 représente un tenseur n dimensionnel. Un dataset se convertit très facilement en numpy array.

Par exemple vous pouvez charger les données samples selon la cellule suivante:

[2]:
```
data_path = './samples.hdf5'

data = h5py.File(data_path , 'r')

signals = np.array(data['signaux'])
snr =  np.array(data['snr'])
labels_id = np.array(data['labels'])

data.close()
```

Vous pouvez récupérer le nom de la correspondance entre un label et le nom du standard d'émetteur correspondant via:

[3]:
```
def get_labels(open_h5_file):
    return {
        open_h5_file['label_name'].attrs[k] : k
        for k in open_h5_file['label_name'].attrs.keys()
    }
```

[4]:
```
with h5py.File(data_path, 'r') as data:
    labels_dict = get_labels(data)

# Associer chaque classe à une liste d'indices correspondants
class_indices = {label: [] for label in labels_dict.values()}
for idx, label_id in enumerate(labels_id):
    class_label = labels_dict.get(label_id, "Unknown")
    class_indices[class_label].append(idx)
```

1.2.1 Visualisation des données

Commencez par étudier les données:

- observez leur taille
- la distribution des différentes classes et des différents SNR dans l'ensemble d'entrainement

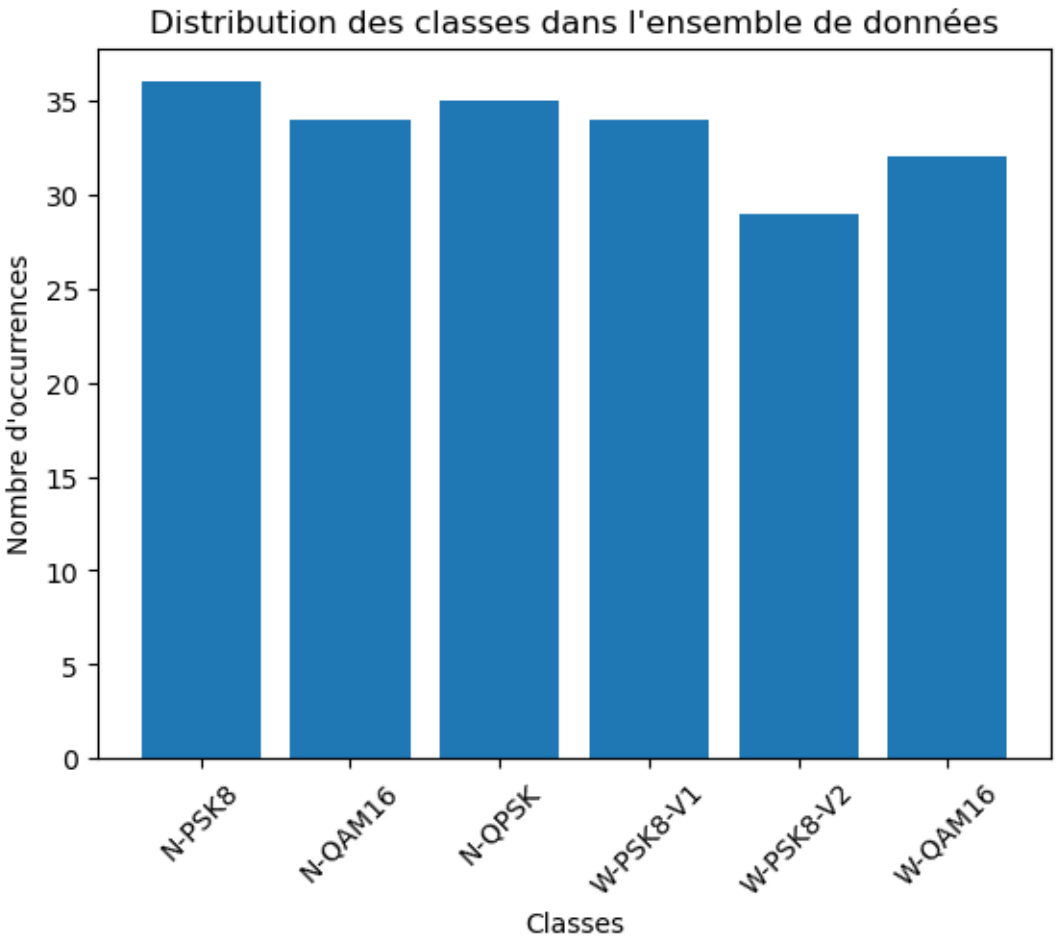
- visualisez quelques signaux bien choisis selon une ou des représentations que vous choisirez

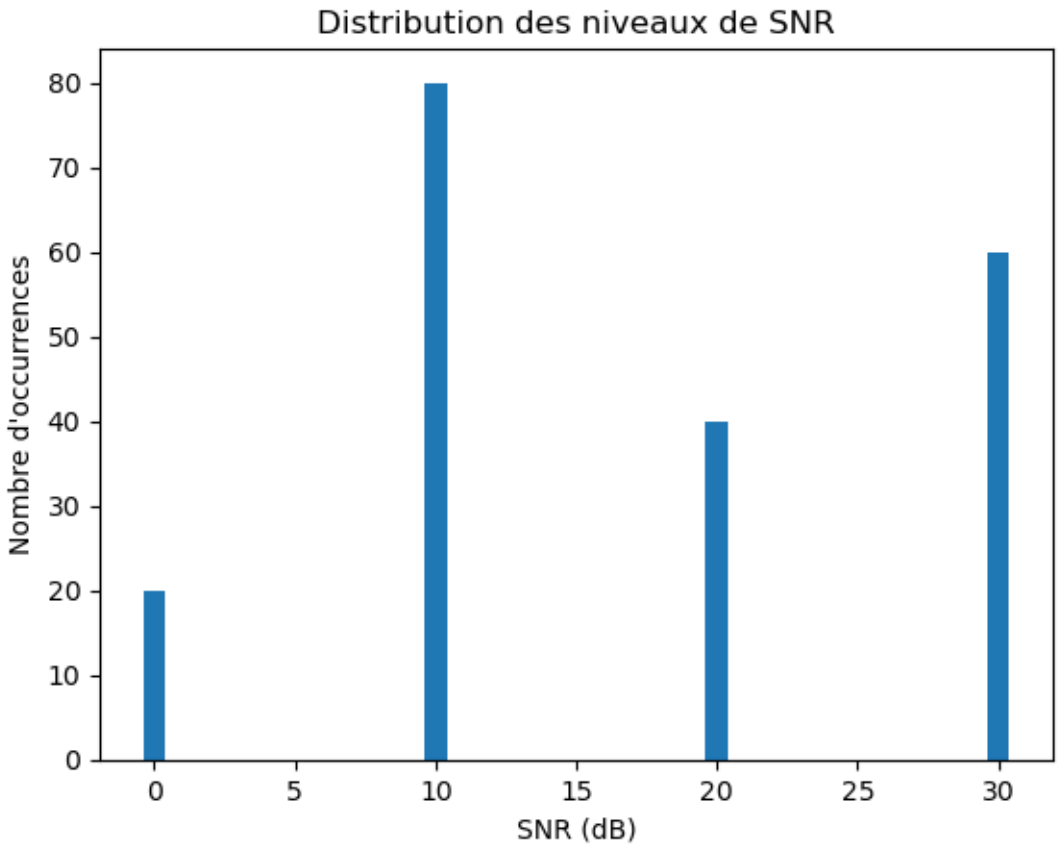
Remarque : dans ce TP il n'y a pas beaucoup à gagner à faire du feature engineering

[5]:
```
# Affichage des dimensions
print("Taille des signaux :", signals.shape)
print("Taille des SNR :", snr.shape)
print("Taille des labels :", labels_id.shape)
```

```
Taille des signaux : (200, 2048, 2)
Taille des SNR : (200,)
Taille des labels : (200,)
```

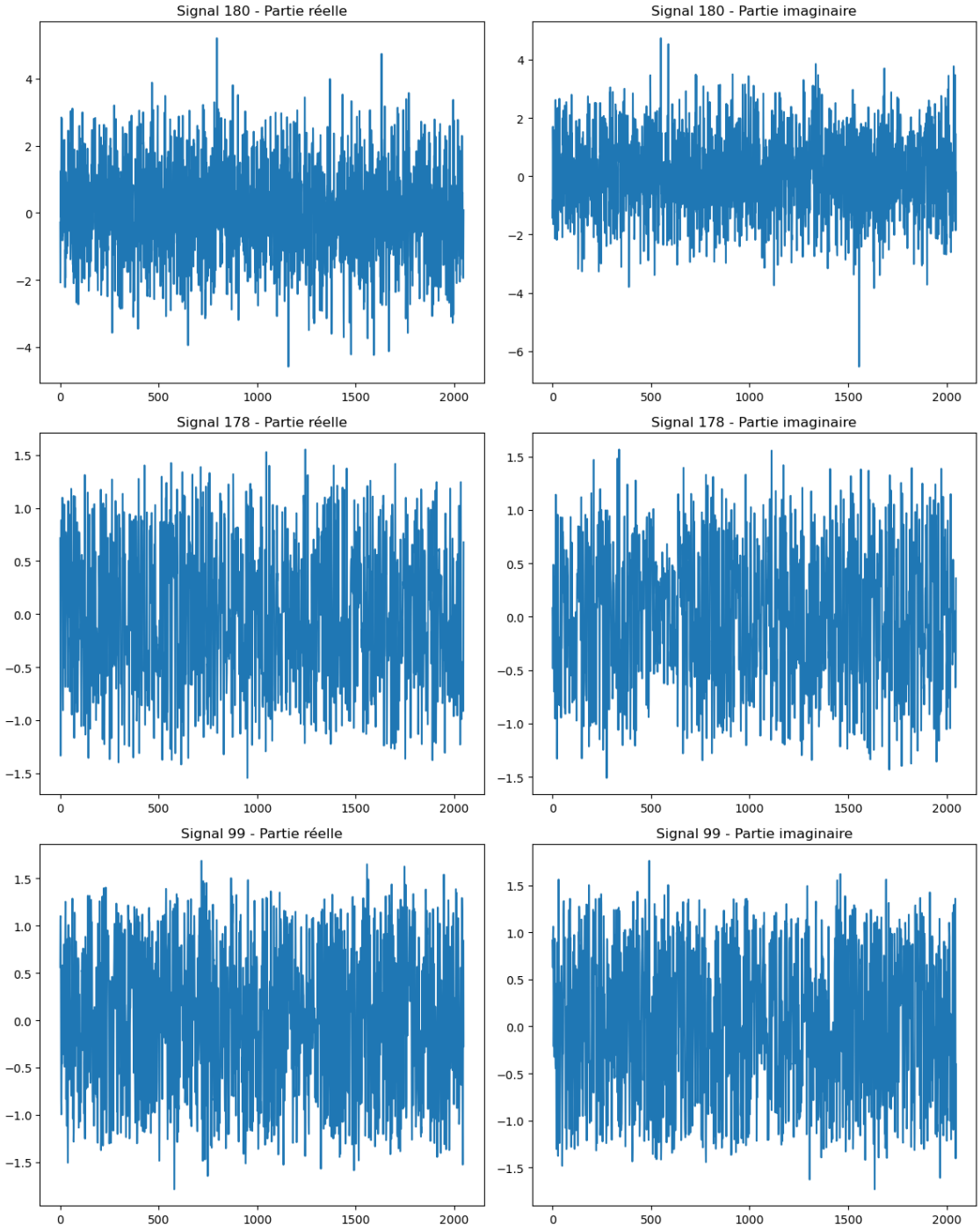
[6]:
```
class_counts = {label: len(indices) for label, indices in class_indices.items()}

# Affichage de la distribution des classes
plt.bar(class_counts.keys(), class_counts.values())
plt.xlabel("Classes")
plt.ylabel("Nombre d'occurrences")
plt.title("Distribution des classes dans l'ensemble de données")
plt.xticks(rotation=45)
plt.show()

# Distribution des SNR
unique_snr, counts_snr = np.unique(snr, return_counts=True)
plt.bar(unique_snr, counts_snr)
plt.title("Distribution des niveaux de SNR")
plt.xlabel("SNR (dB)")
plt.ylabel("Nombre d'occurrences")
plt.show()
```


Distribution des classes dans l'ensemble de données
Nombre d'occurrences
0
5
10
15
20
25
30
35
N-PSK8
N-QAM16
N-QPSK
W-PSK8-V1
W-PSK8-V2
W-QAM16
Classes



Les **6 classes** sont quasiment équiréparties dans notre jeu de données, tandis qu'il y a très clairement une dominance du nombre de signaux avec **un SNR égal à 10**.

[7]:
```
# Visualisation de quelques signaux IQ
fig, axs = plt.subplots(3, 2, figsize=(12, 15))
for i in range(3):
    idx = random.randint(0, signals.shape[0] - 1)
    axs[i, 0].plot(signals[idx, :, 0])  # Partie réelle
    axs[i, 1].plot(signals[idx, :, 1])  # Partie imaginaire
    axs[i, 0].set_title(f'Signal {idx} - Partie réelle')
    axs[i, 1].set_title(f'Signal {idx} - Partie imaginaire')

plt.tight_layout()
plt.show()
```

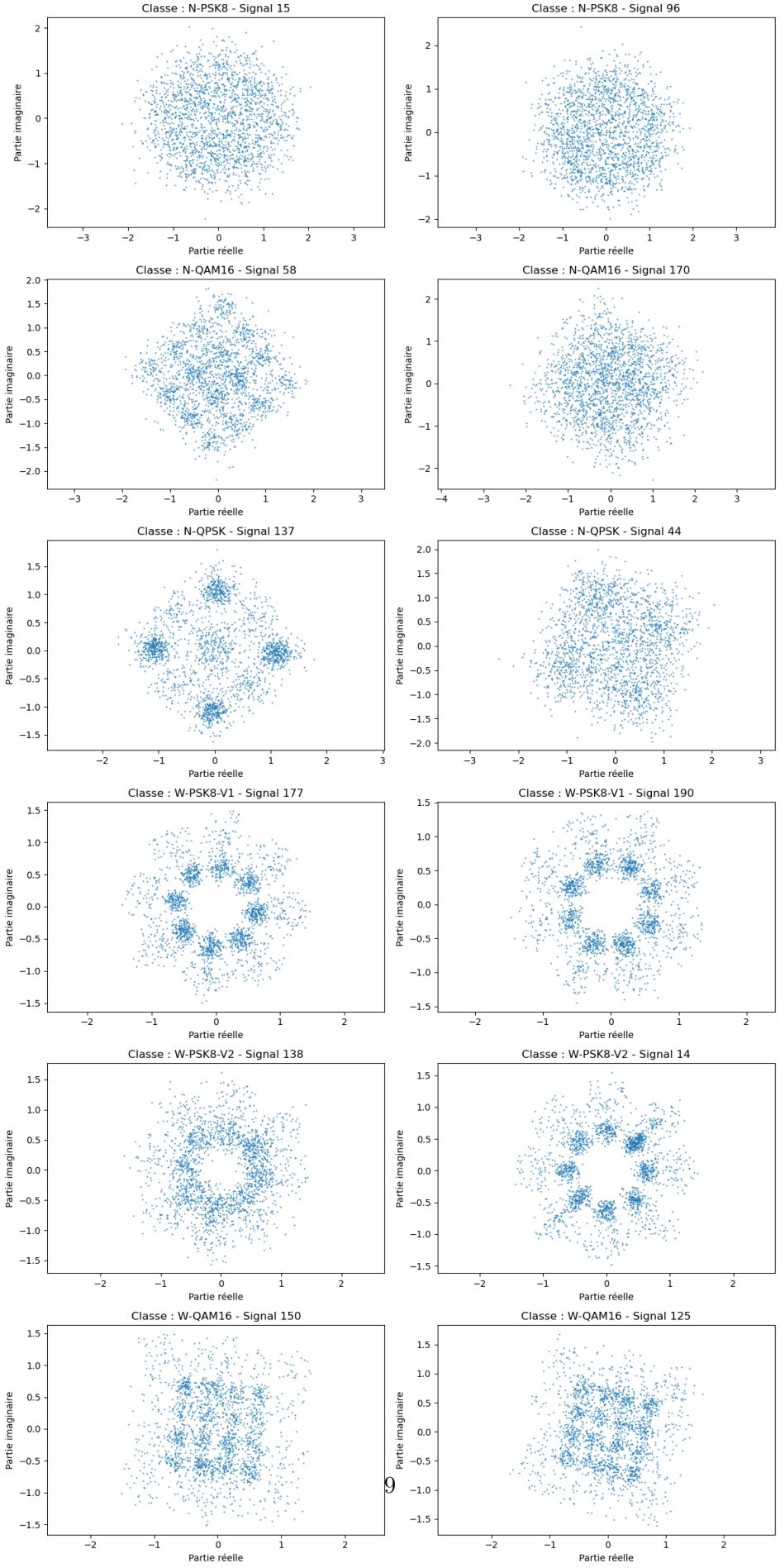


On remarque assez rapidement que cet affichage-là ne va pas beaucoup nous aider dans notre analyse. Il n'y a pas grand chose à en tirer, nous allons donc essayer de les **visualiser directement dans le plan complexe**.

1.2.2 Visualisation des signaux dans le plan complexe

Chaque ligne de cet affichage présente 2 signaux tirés aléatoirement mais **appartenant à une même classe**. Il y a donc 6 lignes, une pour chaque classe.

[8]:
```
random.seed(43)

# Visualisation : une ligne par classe, deux signaux par classe
fig, axs = plt.subplots(6, 2, figsize=(12, 24))
for i, (label, indices) in enumerate(class_indices.items()):
    # Sélectionner deux indices aléatoires pour chaque classe
    selected_indices = random.sample(indices, 2)
    for j, idx in enumerate(selected_indices):
        real_part = signals[idx, :, 0]  # Partie réelle
        imag_part = signals[idx, :, 1]  # Partie imaginaire

        axs[i, j].plot(real_part, imag_part, '.', markersize=1)
        axs[i, j].set_title(f'Classe : {label} - Signal {idx}')
        axs[i, j].set_xlabel("Partie réelle")
        axs[i, j].set_ylabel("Partie imaginaire")
        axs[i, j].axis('equal')  # Échelle égale pour garder la proportion

plt.tight_layout()
plt.show()
```

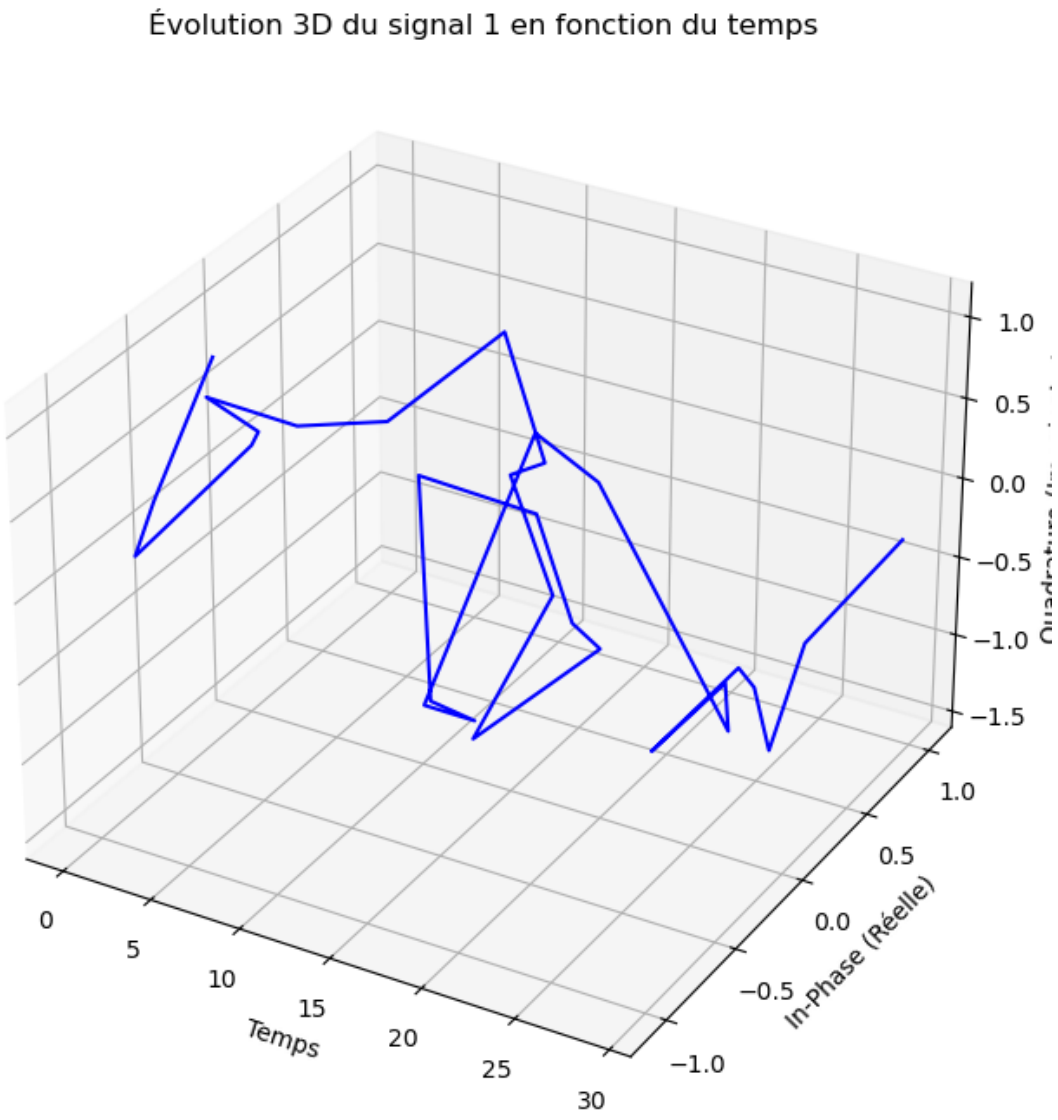

Classe : N-PSK8 - Signal 15
Classe : N-PSK8 - Signal 96
Classe : N-QAM16 - Signal 58
Classe : N-QAM16 - Signal 170
Classe : N-QPSK - Signal 137
Classe : N-QPSK - Signal 44
Classe : W-PSK8-V1 - Signal 177
Classe : W-PSK8-V1 - Signal 190
Classe : W-PSK8-V2 - Signal 138
Classe : W-PSK8-V2 - Signal 14
Classe : W-QAM16 - Signal 150
Classe : W-QAM16 - Signal 125
Partie imaginaire
Partie réelle

L'affichage est très intéressant, mais **il est délicat d'en tirer directement des conclusions**. Certaines classes semblent effectivement avoir des signaux très similaires.

1.2.3 Affichage du début d'un signal en 3 dimensions

On trace les parties réelles et imaginaires en fonction du temps pour un seul signal choisi au hasard.

[9]:
```
# Extraire un seul signal pour le tracé (ex. le premier signal du dataset)
single_signal = signals[0][:30]
time = np.arange(single_signal.shape[0])  # Axe temporel basé sur le nombre␣
↪d'échantillons

# Configuration du tracé 3D
fig = plt.figure(figsize=(8, 16))
ax = fig.add_subplot(111, projection='3d')

# Tracé des parties réelle et imaginaire en fonction du temps
ax.plot(time, single_signal[:, 0], single_signal[:, 1], label="Signal IQ",␣
↪color="b")

# Ajout des étiquettes et du titre
ax.set_title("Évolution 3D du signal 1 en fonction du temps")
ax.set_xlabel("Temps")
ax.set_ylabel("In-Phase (Réelle)")
ax.set_zlabel("Quadrature (Imaginaire)")

plt.show()
```



1.3 Chargement des données en Pytorch

Pour entrainer des réseaux de neurones profond sur nos données nous allons utiliser le framework Pytorch. Une première étape va consister à transférer les données de numpy à PyTorch, cela passe par deux objets : - un Dataset qui modélise le dataset à haut niveau dans la mémoire de l'ordinateur - un Dataloader qui permet d'échantillonner le Dataset Pytorch dans les itérations de l'optimisation du réseau de neurones

Un dataset prend la forme

```
class MyDataset(torch.utils.data.Dataset):

    def __init__(self, path_to_data):
```

```
        ...
    def __len__(self): #retourne le nombre de données dans le dataset
        ...
    def __getitem__(self,i): #retourne pour chaque indice i un couple (data_i, lablel_i), data_i
        ...
```

Implémentez une classe Dataset pour le dataset considéré ici

[10]:
```
class MyDataset(torch.utils.data.Dataset):
    def __init__(self, path_to_data):
        with h5py.File(path_to_data, 'r') as data:
            self.signals = np.array(data['signaux'])
            self.labels = np.array(data['labels'])

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

    def __getitem__(self, i):
        signal = self.signals[i]
        label = self.labels[i]

        # On convertit le signal et le label en tenseurs PyTorch
        signal_tensor = torch.tensor(signal, dtype=torch.float32)
        label_tensor = torch.tensor(label, dtype=torch.long)

        return signal_tensor, label_tensor
```

Instanciez un objet dataset et testez le sur les données samples

```
dataset = MyDataset(...)
```

[11]:
```
dataset = MyDataset('./samples.hdf5')

# On teste le dataset
print("Taille du dataset :", dataset.__len__())

signal, label = dataset.__getitem__(87)
print("Signal shape:", signal.shape)
print("Label:", label)
```

```
Taille du dataset : 200
Signal shape: torch.Size([2048, 2])
Label: tensor(5)
```

Pytorch propose une classe Dataloader qui permet d'échantillonner des batchs de taille fixe à partir d'un dataset. La cellule suivante donne un exemple d'utilisation

[12]:
```
dataloader = DataLoader(dataset,
                        batch_size=10,
                        shuffle=True
```

```
                        )
```

Testez le dataloader pour différentes valeurs de batch_size

[13]:
```
# Test du DataLoader avec différentes valeurs de batch_size
for batch_size in [1, 5, 10, 20]:
    print(f"\nTesting DataLoader with batch_size={batch_size}")

    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

    for batch_idx, (signals, labels) in enumerate(dataloader):
        print(f"Batch {batch_idx + 1}:")
        print("  Signal batch shape:", signals.shape)
        print("  Label batch shape:", labels.shape)

        break
```

```
Testing DataLoader with batch_size=1
Batch 1:
  Signal batch shape: torch.Size([1, 2048, 2])
  Label batch shape: torch.Size([1])

Testing DataLoader with batch_size=5
Batch 1:
  Signal batch shape: torch.Size([5, 2048, 2])
  Label batch shape: torch.Size([5])

Testing DataLoader with batch_size=10
Batch 1:
  Signal batch shape: torch.Size([10, 2048, 2])
  Label batch shape: torch.Size([10])

Testing DataLoader with batch_size=20
Batch 1:
  Signal batch shape: torch.Size([20, 2048, 2])
  Label batch shape: torch.Size([20])
```

1.4 Mise en place d'un réseau "dumb" pour tester la pipeline d'entrainement

Définissez un premier modèle Pytorch qui prend en entrée un batch de données (tenseur de dimensions [B , C, T] avec B la taille du batch, C le nombre de canaux des signaux et T le nombre d'échantillons dans les signaux) et renvoie un batch de vecteur de probabilités (ou de log probabilités si vous préférez) (tenseur de dimensions [B,N] où N est le nombre de classe à identifier).

Ce modèle doit comporter moins de 10000 paramètres libres.

Ce Modèle doit être très simple, il doit être rapide à exécuter, il servira à tester et éventuellement débugger la pipeline d'entrainement que vous mettrez en place un peu plus loin. Un template d'implémentation d'une classe Model se trouve dans les diapositives du cours.

[14]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class DumbModel(nn.Module):
    def __init__(self, num_classes=6, device='cuda'):
        super(DumbModel, self).__init__()
        self.device = device

        # Première couche de convolution avec activation et pooling
        self.ConvLayer_1 = nn.Sequential(
            nn.Conv1d(in_channels=2, out_channels=32, kernel_size=3, stride=1,␣
↪padding=1),  # [B, 32, T]
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=4, stride=4)  # [B, 32, T // 4]
        ).to(self.device)

        # Deuxième couche de convolution avec activation et pooling
        self.ConvLayer_2 = nn.Sequential(
            nn.Conv1d(in_channels=32, out_channels=2, kernel_size=3, stride=1,␣
↪padding=1),  # [B, 2, T // 4]
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=4, stride=4)  # [B, 2, T // 16]
        ).to(self.device)

        self._initialize_fc1()

        # Couches entièrement connectées
        self.fc1 = nn.Linear(self.flattened_size, 32).to(self.device)  # [B, 32]
        self.fc2 = nn.Linear(32, num_classes).to(self.device)  # [B, num_classes]

    def _initialize_fc1(self):
        x = torch.randn(1, 2, 2048).to(self.device)  # Exemple d'entrée :␣
↪[batch=1, in_channels=2, T=2048]
        x = self.ConvLayer_1(x)  # [1, 32, 2048 // 4]
        x = self.ConvLayer_2(x)  # [1, 2, 2048 // 16]
        self.flattened_size = x.view(1, -1).size(1)  # [1, 2 * (2048 // 16)]

    def forward(self, x):
        # On passe les données par les couches de convolution
        x = self.ConvLayer_1(x)  # [B, 32, T // 4]
        x = self.ConvLayer_2(x)  # [B, 2, T // 16]

        x = x.view(x.size(0), -1)  # [B, 2 * (T // 16)]

        # Passer par les couches entièrement connectées
        x = F.relu(self.fc1(x))  # [B, 32]
```

```
        x = self.fc2(x)  # [B, num_classes]
        return F.log_softmax(x, dim=1)
```

Instanciez votre modèle et testez la consistence de ses entrées / sorties vis à vis des données étudiées (test purement fonctionnel, pas besoin de chercher à réaliser un entraînement à ce point).

1.4.1 Premier test avec les données `samples.hdf5`

[15]:
```
print("CUDA disponible :", torch.cuda.is_available())
print("Nom du GPU :", torch.cuda.get_device_name(0) if torch.cuda.is_available()␣
↪else "Aucun GPU détecté")
```

```
CUDA disponible : True
Nom du GPU : NVIDIA GeForce RTX 3050 Ti Laptop GPU
```

[16]:
```
# Initialisation du modèle sur le GPU (ou CPU si indisponible)
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print("Utilisation du dispositif :", device)

model = DumbModel(num_classes=6).to(device)

# On charge un batch de données à partir du dataloader
for sample_input, labels in dataloader:
    # Important : on transfère les données sur le GPU
    sample_input, labels = sample_input.to(device).permute(0, 2, 1), labels.
↪to(device)

    output = model(sample_input)

    # Vérifications et affichages
    print("Taille de l'entrée :", sample_input.shape)  # Devrait être␣
↪[batch_size, 2, 2048]
    print("Taille des labels :", labels.shape)         # Devrait être␣
↪[batch_size]
    print("Taille de la sortie :", output.shape)       # Devrait être␣
↪[batch_size, 6]

    # On affiche les premières log-probabilités pour vérifier
    print("Log-probabilités d'un échantillon :", output[0].detach().cpu().
↪numpy())

    # On ne teste qu'un seul batch, donc on sort de la boucle
    break
```

```
Utilisation du dispositif : cuda
Taille de l'entrée : torch.Size([20, 2, 2048])
Taille des labels : torch.Size([20])
Taille de la sortie : torch.Size([20, 6])
```

```
Log-probabilités d'un échantillon : [-1.9736826 -1.7596016 -1.7120959 -1.8910301
-1.591695  -1.8710707]
```

Estimez par un calcul "théorique" le nombre de paramètres du modèle que vous avez défini et vérifié que le modèle a bien ce nombre de paramètres en pratique par exemple en utilisant la fonction suivante :

[17]:
```python
def count_n_param(model):
    return sum([p.numel() for p in model.parameters()])
```

1.4.2 Calcul Théorique du nombre de Paramètres

1. **Convolution** `ConvLayer_1` :
   - **Poids** : $\text{in\_channels} \times \text{out\_channels} \times \text{kernel\_size} = 2 \times 32 \times 3 = 192$
   - **Biais** : $\text{out\_channels} = 32$
   - **Total pour** `ConvLayer_1` : $192 + 32 = 224$
2. **Convolution** `ConvLayer_2` :
   - **Poids** : $\text{in\_channels} \times \text{out\_channels} \times \text{kernel\_size} = 32 \times 2 \times 3 = 192$
   - **Biais** : $\text{out\_channels} = 2$
   - **Total pour** `ConvLayer_2` : $192 + 2 = 194$
3. **Première Couche Entièrement Connectée** `fc1` :
   - **Poids** : $\text{flattened\_size} \times 32 = 256 \times 32 = 8192$
   - **Biais** : 32
   - **Total pour** `fc1` : $8192 + 32 = 8224$
4. **Deuxième Couche Entièrement Connectée** `fc2` :
   - **Poids** : $32 \times 6 = 192$
   - **Biais** : 6
   - **Total pour** `fc2` : $192 + 6 = 198$

1.4.3 Total Théorique

En additionnant les paramètres des couches :

$$\boxed{224 + 194 + 8224 + 198 = 8840 \text{ paramètres}}$$

[18]:
```python
def count_n_param(model):
    return sum(p.numel() for p in model.parameters())

# Affichage du nombre de paramètres
total_params = count_n_param(model)
print("Nombre total de paramètres (théorique) :", 8840)
print("Nombre total de paramètres (pratique) :", total_params)
```

```
Nombre total de paramètres (théorique) : 8840
Nombre total de paramètres (pratique) : 8840
```

1.5 Mise en place de la pipeline d'entraînement

La pipeline d'entrainement consiste à - charger les données - les batcher - réaliser des itération (epochs) de descente de gradient pour optimiser les paramètres d'un algorithme selon une fonction

de perte (loss) - logger l'évolution au fil des epochs de la loss sur l'ensemble train et l'ensemble de validation et éventuellement de métriques complémentaires

Un cavnevas d'implémentation pourrait être:

```
device = 'cpu' # set so 'cuda:xx' if you have a GPU, xx is GPU index. L'entraînement des réseaux

model = ...  # vous instanciez ici votre modèle

loss = .... # définissez la fonction de perte selon laquelle le modèle sera optimisé

optimizer = torch.optim.Adam(model.parameters()) # en pratique on utilise pas une simple descent

n_epochs = ... # le nombre d'itérations dans l'entrainement

chemin_vers_sauvegarde_model = # chemin vers un fichier où vous sauvegarderez votre modèle après

model.to(device) # on place le modèle dans le GPU si nécessaire

for epoch in range(n_epochs):

    for batch_x,batch_y in dataloader_train:

        batch_x.to(device)
        batch_y.to(device)

        optimizer.zero_grad()

        batch_y_predicted = model(batch_x)

        l = loss(batch_y_predicted, batch_y)
        # loggez la loss sur le batch d'entraînement

        l.backward()

        optimizer.step()

    for batch_x,batch_y in dataloader_valid:

        batch_x.to(device)
        batch_y.to(device)

        with torch.no_grad():
            batch_y_predicted = model(batch_x)

        # loggez la loss et les métriques sur le batch de validation

torch.save(model, chemin_vers_sauvegarde_model)
```

Mettez en place votre pipeline et testez là sur votre modèle dumb. Faites en sorte que votre façon de logger les loss et les métriques vous permette de visualiser l'évolution de ces différents indicateurs sur l'ensemble d'entrainement et de validation au fil des epochs.

Pour simplifier notre travail, nous avons défini deux fonctions : - `train_and_validate` : Elle entraîne et valide un modèle PyTorch, et elle retourne un dictionnaire contenant les pertes et accuracy pour l'entraînement et la validation. - `evaluate_model` : Elle charge le modèle souhaité, et l'évalue sur un ensemble de test, puis affiche les résultats.

[19]:
```
def train_and_validate(model, train_dataloader, valid_dataloader, criterion,␣
↪optimizer, n_epochs, device, save_path):
    train_losses, valid_losses = [], []
    train_accuracies, valid_accuracies = [], []

    for epoch in range(n_epochs):
        # Entraînement
        model.train()
        running_loss = 0.0
        correct_train = 0
        total_train = 0
        for batch_x, batch_y in train_dataloader:
            batch_x, batch_y = batch_x.to(device).permute(0, 2, 1), batch_y.
↪to(device)

            optimizer.zero_grad()
            batch_y_predicted = model(batch_x)
            loss = criterion(batch_y_predicted, batch_y)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(batch_y_predicted, 1)
            correct_train += (predicted == batch_y).sum().item()
            total_train += batch_y.size(0)

        train_loss = running_loss / len(train_dataloader)
        train_accuracy = correct_train / total_train
        train_losses.append(train_loss)
        train_accuracies.append(train_accuracy)

        # Validation
        model.eval()
        running_val_loss = 0.0
        correct_valid = 0
        total_valid = 0
        with torch.no_grad():
            for batch_x, batch_y in valid_dataloader:
```

```
                batch_x, batch_y = batch_x.to(device).permute(0, 2, 1), batch_y.
↪to(device)
                batch_y_predicted = model(batch_x)
                loss = criterion(batch_y_predicted, batch_y)
                running_val_loss += loss.item()
                _, predicted = torch.max(batch_y_predicted, 1)
                correct_valid += (predicted == batch_y).sum().item()
                total_valid += batch_y.size(0)

        valid_loss = running_val_loss / len(valid_dataloader)
        valid_accuracy = correct_valid / total_valid
        valid_losses.append(valid_loss)
        valid_accuracies.append(valid_accuracy)

        print(f"Epoch [{epoch+1}/{n_epochs}], Train Loss: {train_loss:.4f},␣
↪Valid Loss: {valid_loss:.4f}, "
              f"Train Accuracy: {train_accuracy*100:.2f}%, Valid Accuracy:␣
↪{valid_accuracy*100:.2f}%")

    # Sauvegarder le modèle
    torch.save(model.state_dict(), save_path)
    print(f"Modèle sauvegardé à {save_path}")

    return {
        "train_losses": train_losses,
        "valid_losses": valid_losses,
        "train_accuracies": train_accuracies,
        "valid_accuracies": valid_accuracies,
    }

def evaluate_model(model_class, model_path, test_dataloader, device,␣
↪num_classes):
    # On charge le modèle
    model = model_class(num_classes=num_classes, device=device).to(device)
    model.load_state_dict(torch.load(model_path, weights_only=True))  # On␣
↪charge uniquement les poids
    model.eval()
    print(f"Modèle chargé avec succès depuis {model_path} !")

    # Évaluation
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for sample_input, labels in test_dataloader:
            # Important : On transfère les données sur le GPU
            sample_input, labels = sample_input.to(device).permute(0, 2, 1),␣
↪labels.to(device)
```

```
            # On passe les données dans le modèle
            output = model(sample_input)

            # Et hop ! On prédit
            preds = torch.argmax(output, dim=1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    # Accuracy
    accuracy = np.mean(np.array(all_preds) == np.array(all_labels))
    print(f"Test Accuracy: {accuracy * 100:.2f}%")

    # Matrice de confusion
    conf_matrix = confusion_matrix(all_labels, all_preds)
    plt.figure(figsize=(10, 8))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
                xticklabels=[f"Classe {i}" for i in range(num_classes)],
                yticklabels=[f"Classe {i}" for i in range(num_classes)])
    plt.title("Matrice de Confusion")
    plt.xlabel("Classe Prédite")
    plt.ylabel("Classe Réelle")
    plt.show()

    # Rapport de classification
    class_report = classification_report(all_labels, all_preds,␣
↪target_names=[f"Classe {i}" for i in range(num_classes)])
    print("Rapport de classification :")
    print(class_report)

    return {
        "accuracy": accuracy,
        "confusion_matrix": conf_matrix,
        "classification_report": class_report
    }
```

[20]:
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print("Utilisation du dispositif :", device)

# Données
train_dataset = MyDataset('./train.hdf5')
valid_dataset = MyDataset('./validation.hdf5')
test_dataset = MyDataset('./test.hdf5')
train_dataloader = DataLoader(train_dataset, batch_size=10, shuffle=True)
valid_dataloader = DataLoader(valid_dataset, batch_size=10, shuffle=False)
test_dataloader = DataLoader(test_dataset, batch_size=10, shuffle=False)
```

```python
# Modèle
model = DumbModel(num_classes=6).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
n_epochs = 12
chemin_dumb_model = './dumb_model.pth'

# Entraînement et Validation
history = train_and_validate(
    model=model,
    train_dataloader=train_dataloader,
    valid_dataloader=valid_dataloader,
    criterion=criterion,
    optimizer=optimizer,
    n_epochs=n_epochs,
    device=device,
    save_path=chemin_dumb_model,
)

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history["train_losses"], label="Train Loss")
plt.plot(history["valid_losses"], label="Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.title("Évolution de la Loss")

plt.subplot(1, 2, 2)
plt.plot(history["train_accuracies"], label="Train Accuracy")
plt.plot(history["valid_accuracies"], label="Validation Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.title("Évolution de l'Accuracy")

plt.tight_layout()
plt.show()
```

```
Utilisation du dispositif : cuda
Epoch [1/12], Train Loss: 1.2791, Valid Loss: 0.9599, Train Accuracy: 40.61%,
Valid Accuracy: 56.35%
Epoch [2/12], Train Loss: 0.8809, Valid Loss: 0.8103, Train Accuracy: 56.03%,
Valid Accuracy: 56.69%
```

```
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[20], line 21
     18 chemin_dumb_model = './dumb_model.pth'
     20 # Entraînement et Validation
---> 21 history = train_and_validate(
     22     model=model,
     23     train_dataloader=train_dataloader,
     24     valid_dataloader=valid_dataloader,
     25     criterion=criterion,
     26     optimizer=optimizer,
     27     n_epochs=n_epochs,
     28     device=device,
     29     save_path=chemin_dumb_model,
     30 )
     32 plt.figure(figsize=(12, 5))
     33 plt.subplot(1, 2, 1)

Cell In[19], line 22, in train_and_validate(model, train_dataloader,␣
 ↪valid_dataloader, criterion, optimizer, n_epochs, device, save_path)
     20     running_loss += loss.item()
     21     _, predicted = torch.max(batch_y_predicted, 1)
---> 22     correct_train += (predicted == batch_y).sum().item()
     23     total_train += batch_y.size(0)
     25 train_loss = running_loss / len(train_dataloader)

KeyboardInterrupt:
```

Vérifiez que vous avez bien enregistré votre modèle en fin d'entrainement. Chargez le avec la fonction

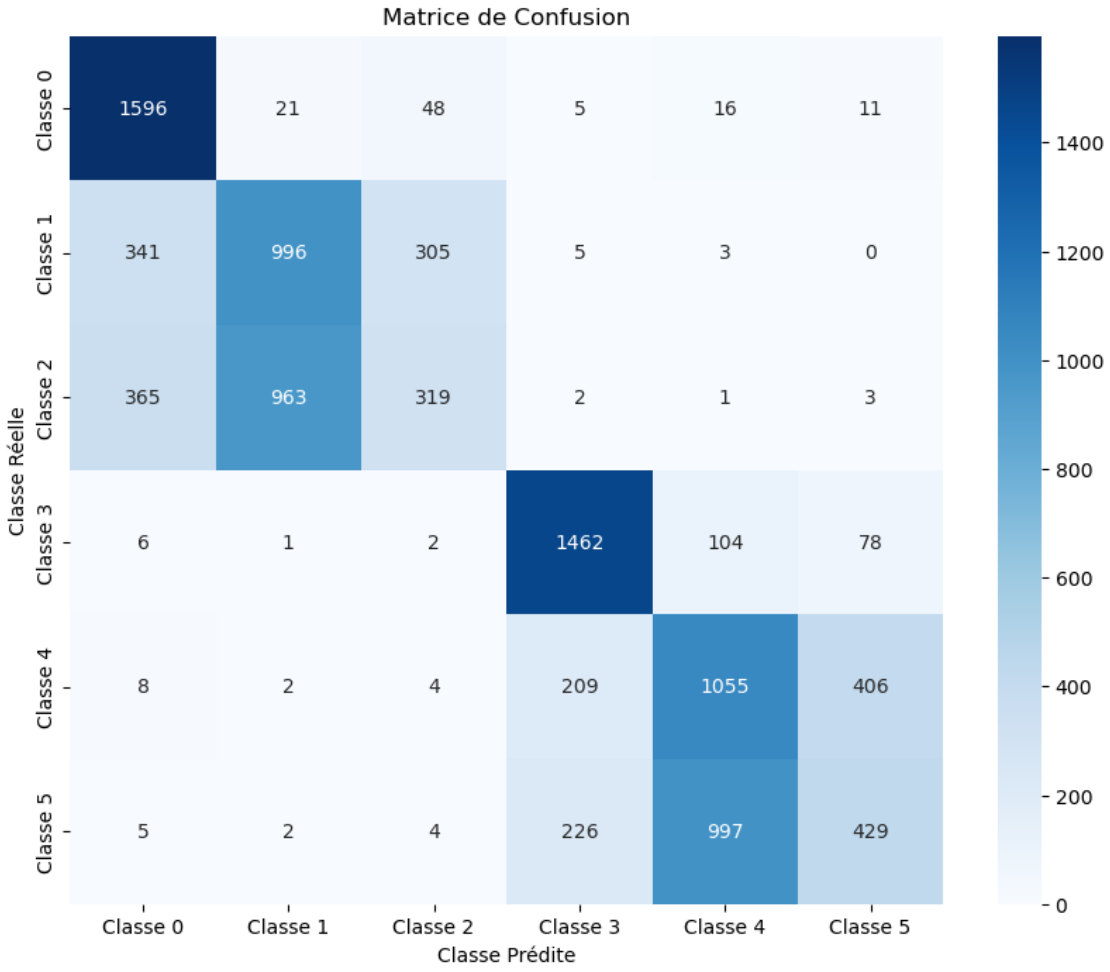
`modele = torch.load(...)`

et vérifiez que vous pouvez l'utiliser sur des données du problème.

[81]:
```
chemin_dumb_model = './dumb_model.pth'  # Chemin du modèle Dumb sauvegardé
device = 'cuda' if torch.cuda.is_available() else 'cpu'
test_dataloader = DataLoader(test_dataset, batch_size=10, shuffle=False)

# On évalue le modèle Dumb
results = evaluate_model(
    model_class=DumbModel,
    model_path=chemin_dumb_model,
    test_dataloader=test_dataloader,
    device=device,
    num_classes=6
)

test_accuracy = results["accuracy"]
conf_matrix = results["confusion_matrix"]
```

```
class_report = results["classification_report"]
```

```
Modèle chargé avec succès depuis ./dumb_model.pth !
Test Accuracy: 58.57%
```




```
Rapport de classification :
              precision    recall  f1-score   support

    Classe 0       0.69      0.94      0.79      1697
    Classe 1       0.50      0.60      0.55      1650
    Classe 2       0.47      0.19      0.27      1653
    Classe 3       0.77      0.88      0.82      1653
    Classe 4       0.48      0.63      0.55      1684
    Classe 5       0.46      0.26      0.33      1663

    accuracy                           0.59     10000
   macro avg       0.56      0.58      0.55     10000
weighted avg       0.56      0.59      0.55     10000
```

1.6 Entraînement de réseaux de neurones

Dans cette partie vous définissez une ou plusieurs architecture de réseaux de neurones profonds et vous les réglez sur les données d'entrainement. Vous pouvez notamment utiliser des réseaux à base de convolutions et/ou de couches réurrentes. Vous pouvez vous inspirer de ce qui a été dit en cours sur la reconnaissance vocale.

Dans un deuxième temps (facultatif), si vous le souhaitez vous pouvez mettre en place des stratégies d'augmentation de données pour améliorer vos résultats. Pour mettre l'augmentation de données en pratique pouvez vous renseigner sur l'argument collate_fn du dataloader standard de Pytorch.

1.6.1 Premier modèle

[75]:
```
class ConvLSTMModel(nn.Module):
    def __init__(self, num_classes=6, device='cuda'):
        super(ConvLSTMModel, self).__init__()
        self.device = device

        # Bloc de convolution 1
        self.ConvBlock1 = nn.Sequential(
            nn.Conv1d(in_channels=2, out_channels=64, kernel_size=3, stride=1,␣
↪padding=1),  # [B, 64, T]
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=4, stride=4)  # [B, 64, T // 4]
        ).to(self.device)

        # Bloc de convolution 2
        self.ConvBlock2 = nn.Sequential(
            nn.Conv1d(in_channels=64, out_channels=128, kernel_size=3, stride=1,␣
↪padding=1),  # [B, 128, T // 4]
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=4, stride=4)  # [B, 128, T // 16]
        ).to(self.device)

        # Réseau LSTM
        self.lstm = nn.LSTM(
            input_size=128,  # Correspond au nombre de canaux après la deuxième␣
↪couche de convolution
            hidden_size=64,
            num_layers=2,
            batch_first=True,
            bidirectional=True
        ).to(self.device)
```

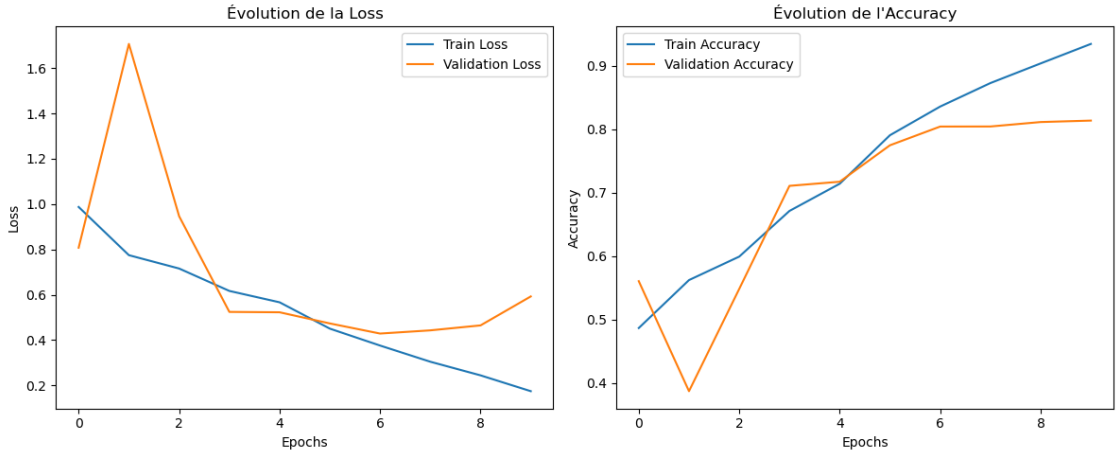
```
        self._initialize_fc1()

        # Couches entièrement connectées
        self.fc1 = nn.Linear(self.flattened_size, 128).to(self.device)
        self.fc2 = nn.Linear(128, num_classes).to(self.device)

    def _initialize_fc1(self):
        x = torch.randn(1, 2, 2048).to(self.device)
        x = self.ConvBlock1(x)  # [1, 64, T // 4]
        x = self.ConvBlock2(x)  # [1, 128, T // 16]

        x = x.permute(0, 2, 1)  # [B, T // 16, 128]
        lstm_out, _ = self.lstm(x)  # [B, T // 16, 128] pour un LSTM␣
↪bidirectionnel
        self.flattened_size = lstm_out.size(1) * lstm_out.size(2)

    def forward(self, x):
        # On passe les données par les blocs de convolution
        x = self.ConvBlock1(x)
        x = self.ConvBlock2(x)

        # On passe ensuite les données dans le LSTM
        x = x.permute(0, 2, 1)
        lstm_out, _ = self.lstm(x)

        x = lstm_out.contiguous().view(lstm_out.size(0), -1)

        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)  # Log-probabilités pour les classes
```

[87]:
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print("Utilisation du dispositif :", device)

# Données
train_dataset = MyDataset('./train.hdf5')
valid_dataset = MyDataset('./validation.hdf5')
test_dataset = MyDataset('./test.hdf5')
train_dataloader = DataLoader(train_dataset, batch_size=10, shuffle=True)
valid_dataloader = DataLoader(valid_dataset, batch_size=10, shuffle=False)
test_dataloader = DataLoader(test_dataset, batch_size=10, shuffle=False)

# Modèle
model = ConvLSTMModel(num_classes=6).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
n_epochs = 10
chemin_model_LSTM = './ConvLSTM_model.pth'

# Entraînement et Validation
history = train_and_validate(
    model=model,
    train_dataloader=train_dataloader,
    valid_dataloader=valid_dataloader,
    criterion=criterion,
    optimizer=optimizer,
    n_epochs=n_epochs,
    device=device,
    save_path=chemin_model_LSTM,
)

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history["train_losses"], label="Train Loss")
plt.plot(history["valid_losses"], label="Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.title("Évolution de la Loss")

plt.subplot(1, 2, 2)
plt.plot(history["train_accuracies"], label="Train Accuracy")
plt.plot(history["valid_accuracies"], label="Validation Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.title("Évolution de l'Accuracy")

plt.tight_layout()
plt.show()
```

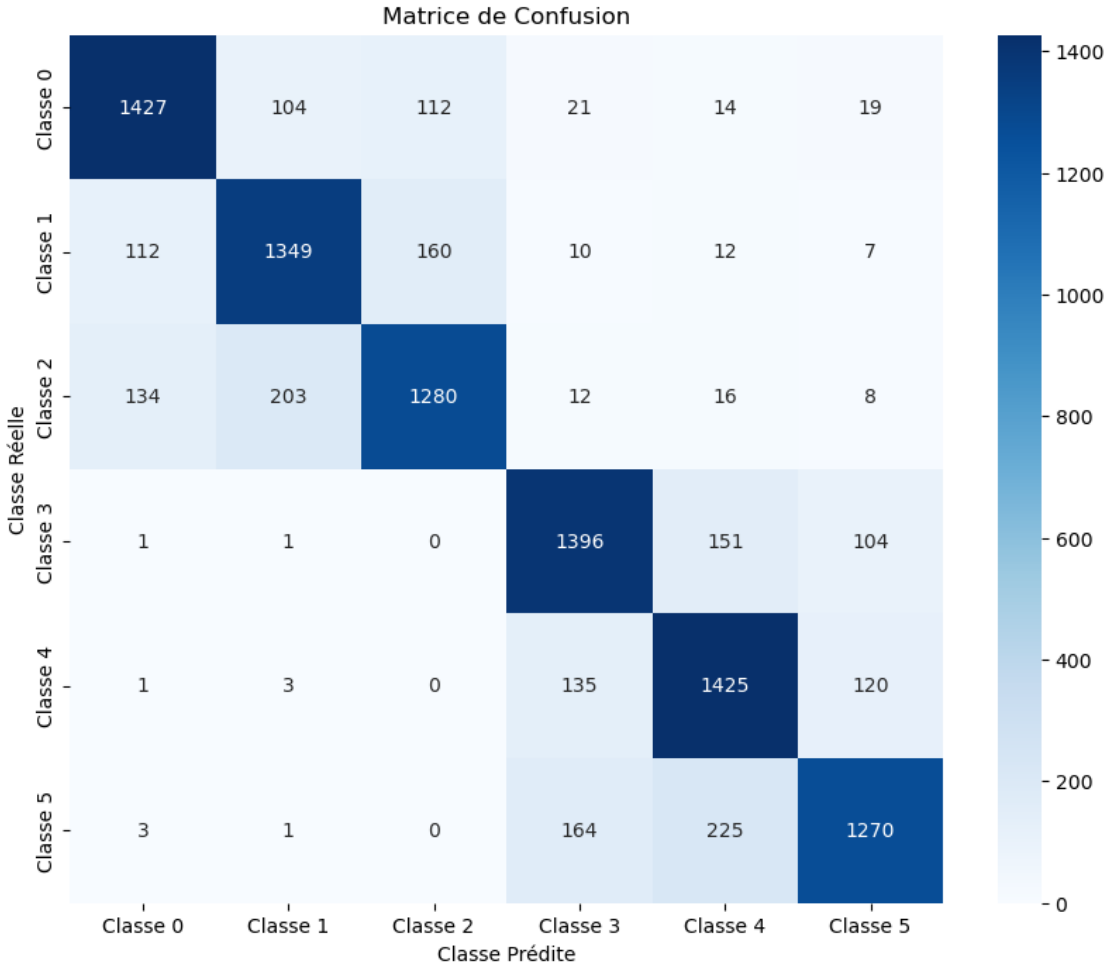
```
Utilisation du dispositif : cuda
Epoch [1/10], Train Loss: 0.9874, Valid Loss: 0.8076, Train Accuracy: 48.69%,
Valid Accuracy: 56.07%
Epoch [2/10], Train Loss: 0.7746, Valid Loss: 1.7070, Train Accuracy: 56.24%,
Valid Accuracy: 38.74%
Epoch [3/10], Train Loss: 0.7157, Valid Loss: 0.9466, Train Accuracy: 59.94%,
Valid Accuracy: 54.85%
Epoch [4/10], Train Loss: 0.6168, Valid Loss: 0.5243, Train Accuracy: 67.13%,
Valid Accuracy: 71.08%
Epoch [5/10], Train Loss: 0.5667, Valid Loss: 0.5227, Train Accuracy: 71.41%,
Valid Accuracy: 71.73%
Epoch [6/10], Train Loss: 0.4505, Valid Loss: 0.4734, Train Accuracy: 79.05%,
Valid Accuracy: 77.47%
```

```
Epoch [7/10], Train Loss: 0.3758, Valid Loss: 0.4287, Train Accuracy: 83.57%,
Valid Accuracy: 80.42%
Epoch [8/10], Train Loss: 0.3044, Valid Loss: 0.4429, Train Accuracy: 87.27%,
Valid Accuracy: 80.43%
Epoch [9/10], Train Loss: 0.2443, Valid Loss: 0.4647, Train Accuracy: 90.36%,
Valid Accuracy: 81.13%
Epoch [10/10], Train Loss: 0.1744, Valid Loss: 0.5925, Train Accuracy: 93.43%,
Valid Accuracy: 81.36%
Modèle sauvegardé à ./ConvLSTM_model.pth
```



[88]:
```python
chemin_model_LSTM = './ConvLSTM_model.pth'
device = 'cuda' if torch.cuda.is_available() else 'cpu'
test_dataloader = DataLoader(test_dataset, batch_size=10, shuffle=False)

# On évalue le modèle
results = evaluate_model(
    model_class=ConvLSTMModel,
    model_path=chemin_model_LSTM,
    test_dataloader=test_dataloader,
    device=device,
    num_classes=6
)

test_accuracy = results["accuracy"]
conf_matrix = results["confusion_matrix"]
class_report = results["classification_report"]
```

```
Modèle chargé avec succès depuis ./ConvLSTM_model.pth !
Test Accuracy: 81.47%
```



Rapport de classification :

	precision	recall	f1-score	support
Classe 0	0.85	0.84	0.85	1697
Classe 1	0.81	0.82	0.81	1650
Classe 2	0.82	0.77	0.80	1653
Classe 3	0.80	0.84	0.82	1653
Classe 4	0.77	0.85	0.81	1684
Classe 5	0.83	0.76	0.80	1663
accuracy			0.81	10000
macro avg	0.82	0.81	0.81	10000
weighted avg	0.82	0.81	0.81	10000

1.6.2 Évaluation du modèle par niveau de SNR

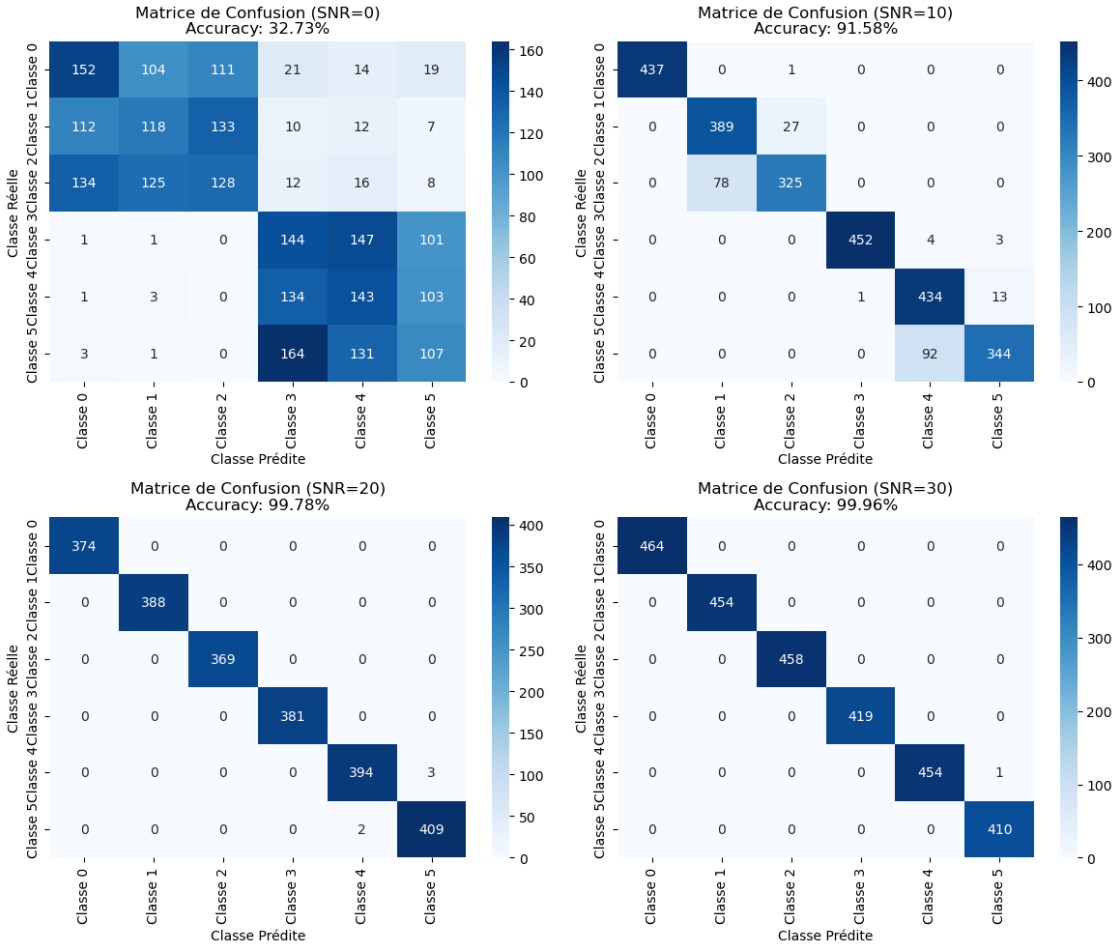
Nous avons obtenu une accuracy de **81.47%**. Ce premier modèle comprenant 2 blocs de convolution et un bloc LSTM semble donc déjà assez performant.

Regardons maintenant cette accuracy en fonction du niveau de bruit des signaux (le SNR) :

[89]:
```
# On charge les SNR depuis les données initiales
def load_snr_from_file(path_to_data):
    with h5py.File(path_to_data, 'r') as data:
        return np.array(data['snr'])

# Charger les SNR associés à l'ensemble de test
snr_test = load_snr_from_file('./test.hdf5')

snr_dict = {0: {"preds": [], "labels": []},
            10: {"preds": [], "labels": []},
            20: {"preds": [], "labels": []},
            30: {"preds": [], "labels": []}}

# On charge le modèle enregistré
modele_charge = ConvLSTMModel(num_classes=6, device=device).to(device)
modele_charge.load_state_dict(torch.load(chemin_model_LSTM, weights_only=True))
modele_charge.eval()
print("Modèle chargé avec succès !")

# Évaluation du modèle
with torch.no_grad():
    for i, (sample_input, labels) in enumerate(test_dataloader):
        sample_input, labels = sample_input.to(device).permute(0, 2, 1), labels.
 ↪to(device)

        output = modele_charge(sample_input)
        preds = torch.argmax(output, dim=1).cpu().numpy()
        labels = labels.cpu().numpy()  # Labels réels

        # Comme nous avons mis "shuffle=False" dans les données tests, on peut␣
 ↪associer les SNR correspondants aux prédictions
        batch_snr = snr_test[i * test_dataloader.batch_size : (i + 1) *␣
 ↪test_dataloader.batch_size]

        for s, p, l in zip(batch_snr, preds, labels):
            snr_dict[int(s)]["preds"].append(p)
            snr_dict[int(s)]["labels"].append(l)

# On affiche des matrices de confusion pour chaque SNR et calcul des accuracy :
plt.figure(figsize=(12, 10))
accuracies = {}
```

```
for idx, (snr_level, data) in enumerate(snr_dict.items()):
    preds = np.array(data["preds"])
    labels = np.array(data["labels"])

    conf_matrix = confusion_matrix(labels, preds)

    accuracy = np.mean(preds == labels)
    accuracies[snr_level] = accuracy

    plt.subplot(2, 2, idx + 1)
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',␣
↪xticklabels=[f"Classe {i}" for i in range(6)],
                yticklabels=[f"Classe {i}" for i in range(6)])
    plt.title(f"Matrice de Confusion (SNR={snr_level})\nAccuracy: {accuracy*100:.
↪2f}%")
    plt.xlabel("Classe Prédite")
    plt.ylabel("Classe Réelle")

plt.tight_layout()
plt.show()

print("Accuracies par niveau de SNR :")
for snr_level, acc in accuracies.items():
    print(f"SNR = {snr_level} : Accuracy = {acc*100:.2f}%")
```

```
Modèle chargé avec succès !
```




```
Accuracies par niveau de SNR :
SNR = 0 : Accuracy = 32.73%
SNR = 10 : Accuracy = 91.58%
SNR = 20 : Accuracy = 99.78%
SNR = 30 : Accuracy = 99.96%
```

1.6.3 Deuxième modèle

[ ]:
```python
class ConvLSTMModel2(nn.Module):
    def __init__(self, num_classes=6, device='cuda'):
        super(ConvLSTMModel2, self).__init__()
        self.device = device

        # Bloc de convolution 1
        self.ConvBlock1 = nn.Sequential(
            nn.Conv1d(in_channels=2, out_channels=64, kernel_size=3, stride=1,␣
↪padding=1),  # [B, 64, T]
            nn.BatchNorm1d(64),
```

```
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=4, stride=4)  # [B, 64, T // 4]
        ).to(self.device)

        # Bloc de convolution 2
        self.ConvBlock2 = nn.Sequential(
            nn.Conv1d(in_channels=64, out_channels=128, kernel_size=3, stride=1,␣
↪padding=1),  # [B, 128, T // 4]
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=4, stride=4)  # [B, 128, T // 16]
        ).to(self.device)

        # Ajout d'un 3ème bloc de convolution
        self.ConvBlock3 = nn.Sequential(
            nn.Conv1d(in_channels=128, out_channels=256, kernel_size=3,␣
↪stride=1, padding=1),  # [B, 256, T // 16]
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=4, stride=4)  # [B, 256, T // 64]
        ).to(self.device)

        # Réseau LSTM
        self.lstm = nn.LSTM(
            input_size=256,
            hidden_size=64,
            num_layers=2,
            batch_first=True,
            bidirectional=True
        ).to(self.device)

        self._initialize_fc1()

        # Couches entièrement connectées
        self.fc1 = nn.Linear(self.flattened_size, 128).to(self.device)
        self.fc2 = nn.Linear(128, num_classes).to(self.device)

    def _initialize_fc1(self):
        x = torch.randn(1, 2, 2048).to(self.device)
        x = self.ConvBlock1(x)  # [1, 64, T // 4]
        x = self.ConvBlock2(x)  # [1, 128, T // 16]
        x = self.ConvBlock3(x)  # [1, 256, T // 64]

        x = x.permute(0, 2, 1)  # On permute pour correspondre à [B, T // 64,␣
↪256]
        lstm_out, _ = self.lstm(x)  # [B, T // 64, 128]
        self.flattened_size = lstm_out.size(1) * lstm_out.size(2)
```

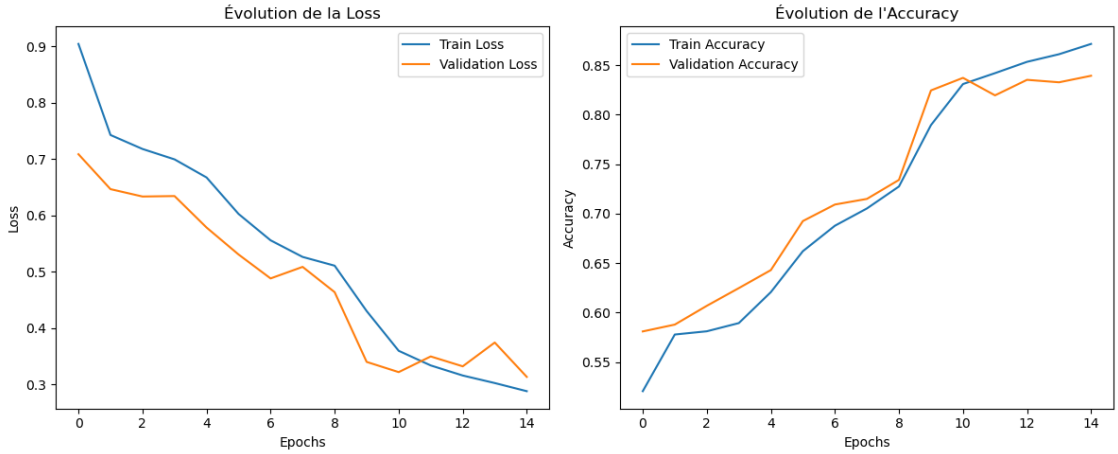
```
    def forward(self, x):
        # On passe les données par les blocs de convolution
        x = self.ConvBlock1(x)
        x = self.ConvBlock2(x)
        x = self.ConvBlock3(x)

        # Passer les données dans le LSTM
        x = x.permute(0, 2, 1)
        lstm_out, _ = self.lstm(x)

        x = lstm_out.contiguous().view(lstm_out.size(0), -1)

        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)  # Log-probabilités pour les classes
```

[ ]:
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print("Utilisation du dispositif :", device)

train_dataset = MyDataset('./train.hdf5')
valid_dataset = MyDataset('./validation.hdf5')
test_dataset = MyDataset('./test.hdf5')
train_dataloader = DataLoader(train_dataset, batch_size=10, shuffle=True)
valid_dataloader = DataLoader(valid_dataset, batch_size=10, shuffle=False)
test_dataloader = DataLoader(test_dataset, batch_size=10, shuffle=False)

# Modèle
model = ConvLSTMModel2(num_classes=6).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
n_epochs = 15
chemin_model_LSTM2 = './ConvLSTM_model2.pth'

# Entraînement et Validation
history = train_and_validate(
    model=model,
    train_dataloader=train_dataloader,
    valid_dataloader=valid_dataloader,
    criterion=criterion,
    optimizer=optimizer,
    n_epochs=n_epochs,
    device=device,
    save_path=chemin_model_LSTM2,
)
```

```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history["train_losses"], label="Train Loss")
plt.plot(history["valid_losses"], label="Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.title("Évolution de la Loss")

plt.subplot(1, 2, 2)
plt.plot(history["train_accuracies"], label="Train Accuracy")
plt.plot(history["valid_accuracies"], label="Validation Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.title("Évolution de l'Accuracy")

plt.tight_layout()
plt.show()
```

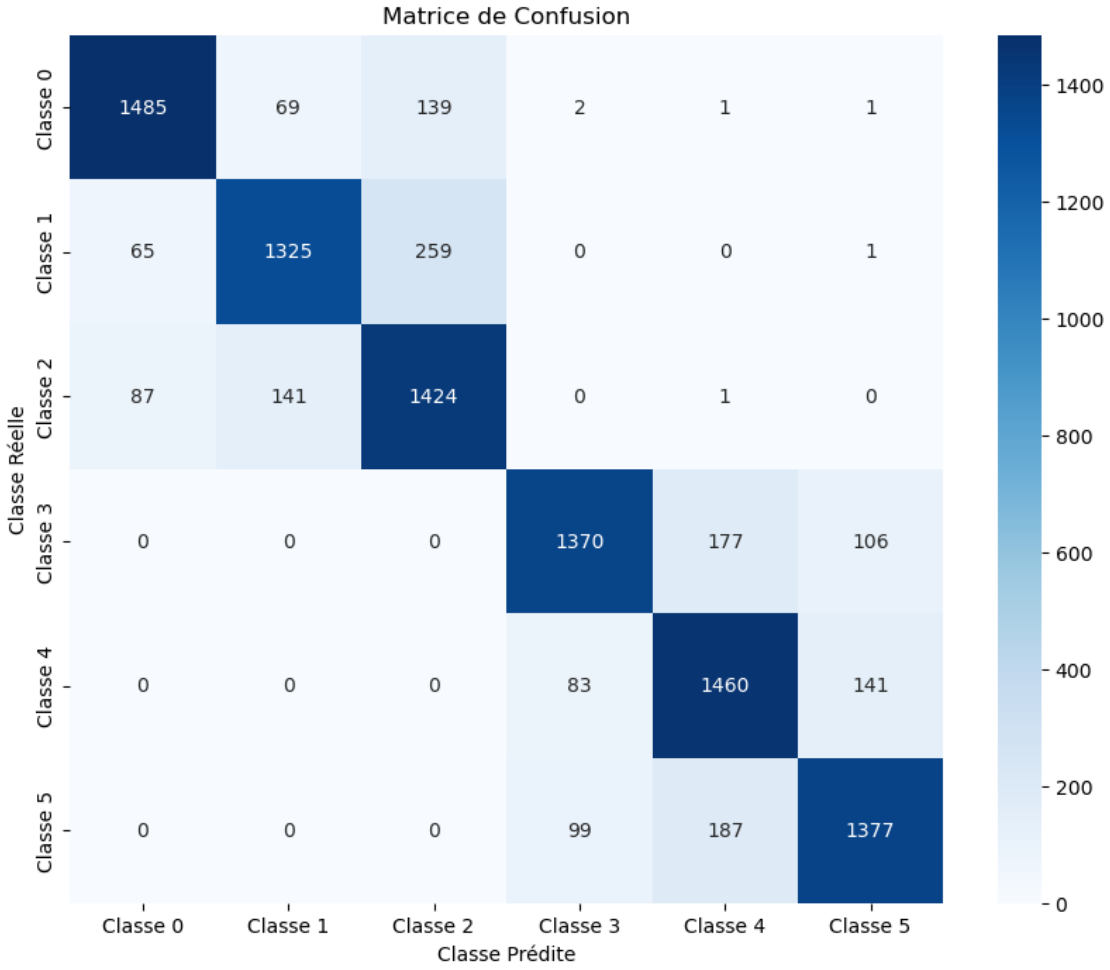
```
Utilisation du dispositif : cuda
Epoch [1/15], Train Loss: 0.9042, Valid Loss: 0.7084, Train Accuracy: 52.05%,
Valid Accuracy: 58.09%
Epoch [2/15], Train Loss: 0.7425, Valid Loss: 0.6464, Train Accuracy: 57.78%,
Valid Accuracy: 58.78%
Epoch [3/15], Train Loss: 0.7177, Valid Loss: 0.6333, Train Accuracy: 58.10%,
Valid Accuracy: 60.68%
Epoch [4/15], Train Loss: 0.6992, Valid Loss: 0.6341, Train Accuracy: 58.93%,
Valid Accuracy: 62.47%
Epoch [5/15], Train Loss: 0.6671, Valid Loss: 0.5783, Train Accuracy: 62.06%,
Valid Accuracy: 64.29%
Epoch [6/15], Train Loss: 0.6024, Valid Loss: 0.5304, Train Accuracy: 66.20%,
Valid Accuracy: 69.25%
Epoch [7/15], Train Loss: 0.5555, Valid Loss: 0.4879, Train Accuracy: 68.78%,
Valid Accuracy: 70.92%
Epoch [8/15], Train Loss: 0.5261, Valid Loss: 0.5084, Train Accuracy: 70.52%,
Valid Accuracy: 71.49%
Epoch [9/15], Train Loss: 0.5105, Valid Loss: 0.4635, Train Accuracy: 72.74%,
Valid Accuracy: 73.41%
Epoch [10/15], Train Loss: 0.4299, Valid Loss: 0.3397, Train Accuracy: 78.95%,
Valid Accuracy: 82.46%
Epoch [11/15], Train Loss: 0.3594, Valid Loss: 0.3216, Train Accuracy: 83.10%,
Valid Accuracy: 83.73%
Epoch [12/15], Train Loss: 0.3334, Valid Loss: 0.3494, Train Accuracy: 84.21%,
Valid Accuracy: 81.96%
Epoch [13/15], Train Loss: 0.3156, Valid Loss: 0.3319, Train Accuracy: 85.35%,
Valid Accuracy: 83.53%
Epoch [14/15], Train Loss: 0.3022, Valid Loss: 0.3741, Train Accuracy: 86.12%,
```

```
Valid Accuracy: 83.28%
Epoch [15/15], Train Loss: 0.2877, Valid Loss: 0.3131, Train Accuracy: 87.16%,
Valid Accuracy: 83.94%
Modèle sauvegardé à ./ConvLSTM_model2.pth
```



[86]:
```python
chemin_model_LSTM2 = './ConvLSTM_model2.pth'
device = 'cuda' if torch.cuda.is_available() else 'cpu'
test_dataloader = DataLoader(test_dataset, batch_size=10, shuffle=False)

# On évalue le modèle
results = evaluate_model(
    model_class=ConvLSTMModel2,
    model_path=chemin_model_LSTM2,
    test_dataloader=test_dataloader,
    device=device,
    num_classes=6
)

test_accuracy = results["accuracy"]
conf_matrix = results["confusion_matrix"]
class_report = results["classification_report"]
```

```
Modèle chargé avec succès depuis ./ConvLSTM_model2.pth !
Test Accuracy: 84.41%
```



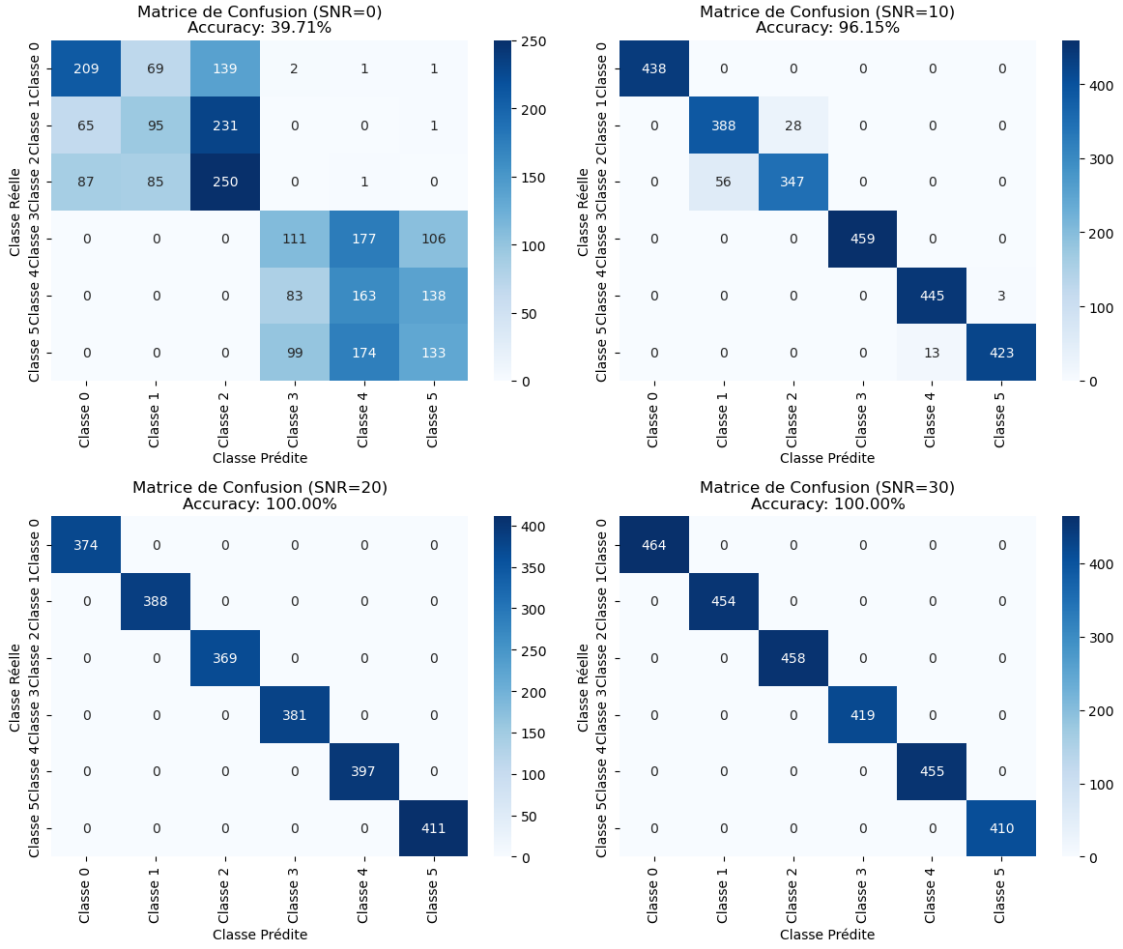

```
Rapport de classification :
              precision    recall  f1-score   support

    Classe 0       0.91      0.88      0.89      1697
    Classe 1       0.86      0.80      0.83      1650
    Classe 2       0.78      0.86      0.82      1653
    Classe 3       0.88      0.83      0.85      1653
    Classe 4       0.80      0.87      0.83      1684
    Classe 5       0.85      0.83      0.84      1663

    accuracy                           0.84     10000
   macro avg       0.85      0.84      0.84     10000
weighted avg       0.85      0.84      0.84     10000
```

[ ]:
```python
def load_snr_from_file(path_to_data):
    with h5py.File(path_to_data, 'r') as data:
```

```
        return np.array(data['snr'])

snr_test = load_snr_from_file('./test.hdf5')

snr_dict = {0: {"preds": [], "labels": []},
            10: {"preds": [], "labels": []},
            20: {"preds": [], "labels": []},
            30: {"preds": [], "labels": []}}

modele_charge = ConvLSTMModel2(num_classes=6, device=device).to(device)
modele_charge.load_state_dict(torch.load(chemin_model_LSTM2, weights_only=True))
modele_charge.eval()
print("Modèle chargé avec succès !")

with torch.no_grad():
    for i, (sample_input, labels) in enumerate(test_dataloader):
        sample_input, labels = sample_input.to(device).permute(0, 2, 1), labels.
 ↪to(device)

        output = modele_charge(sample_input)
        preds = torch.argmax(output, dim=1).cpu().numpy()
        labels = labels.cpu().numpy()

        batch_snr = snr_test[i * test_dataloader.batch_size : (i + 1) *␣
 ↪test_dataloader.batch_size]

        for s, p, l in zip(batch_snr, preds, labels):
            snr_dict[int(s)]["preds"].append(p)
            snr_dict[int(s)]["labels"].append(l)

plt.figure(figsize=(12, 10))

accuracies = {}

for idx, (snr_level, data) in enumerate(snr_dict.items()):
    preds = np.array(data["preds"])
    labels = np.array(data["labels"])

    conf_matrix = confusion_matrix(labels, preds)

    accuracy = np.mean(preds == labels)
    accuracies[snr_level] = accuracy

    plt.subplot(2, 2, idx + 1)
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',␣
 ↪xticklabels=[f"Classe {i}" for i in range(6)],
                yticklabels=[f"Classe {i}" for i in range(6)])
```

```
    plt.title(f"Matrice de Confusion (SNR={snr_level})\nAccuracy: {accuracy*100:.
↪2f}%")
    plt.xlabel("Classe Prédite")
    plt.ylabel("Classe Réelle")

plt.tight_layout()
plt.show()

print("Accuracies par niveau de SNR :")
for snr_level, acc in accuracies.items():
    print(f"SNR = {snr_level} : Accuracy = {acc*100:.2f}%")
```

```
Modèle chargé avec succès !
```




```
Accuracies par niveau de SNR :
SNR = 0 : Accuracy = 39.71%
SNR = 10 : Accuracy = 96.15%
SNR = 20 : Accuracy = 100.00%
```

```
SNR = 30 : Accuracy = 100.00%
```

1.7 Synthèse de résultats

Une fois que votre ou vos réseaux sont entrainez vous comparez leurs performances selon les métriques définies en introduction sur l'ensemble de test sans oublier de mesurer également la complexité de chaque approche en termes de nombre de paramètres. Si vous avez testé des approches qui vous semblent avoir échoué vous pouvez rédiger un petit paragraphe pour expliquer votre analyse de cet échec.

1.7.1 Découverte de Torch

Nous avons commencé notre travail en découvrant **Torch**. En effet, nous sommes tous deux des débutants en **Deep Learning**, et toute la partie facultative du cours nous a été d'une grande aide pour comprendre ce domaine ainsi que l'implémentation des réseaux de neurones.

1.7.2 Modèle 1 : Dumb Model (Baseline)

Après ces premières étapes de compréhension, nous avons développé un modèle **Dumb**, qui sert de baseline. Ce modèle simple, constitué de deux couches de convolution suivies de deux couches entièrement connectées, nous a permis d'évaluer une première performance brute sur les données. Ce modèle a atteint une **accuracy de 58.57%** sur les données de test. Bien qu'il soit limité, ce modèle nous a fourni un **point de comparaison** pour évaluer les améliorations des modèles plus complexes.

1.7.3 Modèle 2 : ConvLSTM1

Après le Dumb Model, nous avons introduit une couche récurrente (**LSTM**) dans le modèle, le nommant **ConvLSTM1**. Avec deux couches de convolution suivies d'un LSTM bidirectionnel, ce modèle avait pour but de capturer les relations temporelles dans les signaux. Nous avons obtenu des résultats significatifs : - **Accuracy globale sur le test** : **81.47%** - **Accuracies par niveau de SNR** : - **SNR = 0** : Accuracy = **32.73%** - **SNR = 10** : Accuracy = **91.58%** - **SNR = 20** : Accuracy = **99.78%** - **SNR = 30** : Accuracy = **99.96%**

Nous sommes déjà très satisfait des résultats de ce permier modèle utilisant un LSTM, puisque l'accuracy globale est très bonne, et on atteint quasiment **100%** d'accuracy pour les signaux avec des SNR de 20 ou 30.

1.7.4 Modèle 3 : ConvLSTM2

Nous avons ensuite décidé d'enrichir davantage l'extraction des caractéristiques avec une troisième couche convolutive, donnant naissance au modèle **ConvLSTM2**. Ce modèle a montré des améliorations supplémentaires, surtout pour **les SNR faibles** : - **Accuracy globale sur le test** : **84.41%** - **Accuracies par niveau de SNR** : - **SNR = 0** : Accuracy = **39.71%** - **SNR = 10** : Accuracy = **96.15%** - **SNR = 20** : Accuracy = **100.00%** - **SNR = 30** : Accuracy = **100.00%**

Les résultats sont vraiment très bons. Pour les signaux avec des niveaux de SNR de 10, 20 ou 30, le modèle fournit des résultats presque parfait (avec **des accuracy au dessus de 96%**).

Pour les signaux avec énormément de bruit (**SNR = 0**), les résultats sont forcément moins bon, mais on arrive tout de même à en tirer quelque chose. En effet, nous avons réussi à augmenter cette accuracy jusqu'à **preque 40%**, ce qui est déjà plutôt intéressant.

1.7.5 Tests non concluants

Tout au long de ce projet, nous avons mené de nombreuses expériences pour optimiser nos modèles. Certaines ont abouti à des échecs ou des résultats décevants : - **Ajustement du learning rate** : Nous avons testé plusieurs valeurs (au-dessus et en dessous de 0.001), mais les performances sont restées stables. - **Modification de la taille des batchs** : Bien que cela ait impacté le temps d'entraînement, les résultats finaux ont peu changé. - **Augmentation de la profondeur du LSTM** : En ajoutant plus de couches au LSTM, nous avons observé une dégradation des performances, avec une accuracy chutant à **60%**. - **Ajout de Dropout** : La régularisation introduite par Dropout a fortement réduit l'accuracy (moins de **60%**), probablement en raison d'une perte excessive d'informations dans nos modèles déjà peu complexes. - **Changement du nombre de neuronnes par couches convolutives** : Nous avons réalisés beaucoup d'essais avec différentes valeurs, et nous sommes restés sur des valeurs qui semblaient fournir les meilleurs résultats.

[91]:
```
model = DumbModel(num_classes=6).to(device)
total_params = count_n_param(model)
print("Nombre total de paramètres (pratique) :", total_params)

model = ConvLSTMModel(num_classes=6).to(device)
total_params = count_n_param(model)
print("Nombre total de paramètres (pratique) :", total_params)

model = ConvLSTMModel2(num_classes=6).to(device)
total_params = count_n_param(model)
print("Nombre total de paramètres (pratique) :", total_params)
```

```
Nombre total de paramètres (pratique) : 8840
Nombre total de paramètres (pratique) : 2322246
Nombre total de paramètres (pratique) : 913990
```

1.7.6 Comparaison des Modèles selon leur complexité

Modèle	Accuracy Globale (%)	Nombre de Paramètres	Commentaire
Dumb Model	58.57	8 840	Modèle baseline
ConvLSTM1	81.47	2 322 246	Modèle avec 2 Convolutions + 1 LSTM
ConvLSTM2	84.41	913 990	Modèle avec 3 Convolutions + 1 LSTM

Pour conclure, ce dernier modèle possède relativement **moins de paramètres que le modèle**

numéro 2, mais arrive à un niveau d'accuracy vraiment très bon. Il est donc **moins complexe**, et **très performant**.