Apprentissage_ECG_20252026

December 23, 2025

1 Classification des ECG de la base de données PTB-XL

Ce devoir repose sur le jeu de données PTB-XL disponible ici. Il est indispenable de télécharger le jeu de données (∼ 3Go) et le renommer **ptbxl**.

L'objectif de ce travail est d'entraîner un modèle de classification d'ECG en 5 classes décrites au milieu de la section *Data Description* de la pageweb du jeu de données.

- NORM Normal ECG
- MI Myocardial Infarction
- STTC ST/T Change
- CD Conduction Disturbance
- HYP Hypertrophy

Un ecg est composé de 12 dérivations (ou **leads** en anglais) échantillonnés à une fréquence 100hz (ou 500 hz dans le jeu de données) sur 10 secondes. Nous avons donc 12 dérivations de taille 1000 pour chaque ECG.

1.1 Chargement des librairies

Nous allons avoir besoin des librairies suivantes à installer si elles ne le sont pas.

[ ]:
```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import wfdb
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MultiLabelBinarizer
from sklearn.metrics import classification_report, confusion_matrix,␣
↪roc_auc_score
import seaborn as sns

# Deep Learning imports
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

try:
```

```
    from iterstrat.ml_stratifiers import MultilabelStratifiedShuffleSplit
    STRATIFIED_AVAILABLE = True
except ImportError:
    print("Attention: iterative-stratification  n'est pas installé. Utilisation␣
 ↪d'un regular split.")
    print("Installer avec: pip install iterative-stratification")
    STRATIFIED_AVAILABLE = False
```

1.2 Fonction de chargement des données

Afin de se consacrer pleinement à la mise en place du modèle d'apprentissage, la fonction suivante permet de charger les données à partir du répertoire **ptbxl**.

[ ]:
```
def load_ptbxl_data(data_path='ptbxl/', sampling_rate=100):
    """
    chargement dataset PTB-XL
    Args:
        data_path: chemin vers le dossier PTB-XL
        sampling_rate: frequence d'echantillonnage 100 or 500 Hz
    Returns:
        X: ECG signals, Y: labels, metadata
    """
    # Load metadata
    metadata_file = os.path.join(data_path, 'ptbxl_database.csv')
    print(metadata_file)
    Y = pd.read_csv(metadata_file, index_col='ecg_id')
    Y.scp_codes = Y.scp_codes.apply(lambda x: eval(x))

    # Load signals
    if sampling_rate == 100:
        data_folder = os.path.join(data_path, 'records100')
    else:
        data_folder = os.path.join(data_path, 'records500')

    X = []
    valid_indices = []

    print("Chargement des ECG ...")
    for idx, row in Y.iterrows():
        file_path = os.path.join(data_path, row.filename_lr if sampling_rate ==␣
 ↪100 else row.filename_hr)
        try:
            signal, _ = wfdb.rdsamp(file_path)
            X.append(signal)
            valid_indices.append(idx)
        except Exception as e:
            print(f"Error loading {file_path}: {e}")
```

```
            continue

    X = np.array(X)
    Y = Y.loc[valid_indices]

    print(f"chargés {len(X)} enregistrements ECG")
    print(f"Dimensions du signal (ecg, temps, derivation): {X.shape}")  #␣
↪(num_samples, time_steps, num_leads)

    return X, Y
```

1.3 Déclaration de variables globales

[ ]:
```
# Calcul sur GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# Afficher les infos GPU si disponibles
if torch.cuda.is_available():
   print(f"GPU Name: {torch.cuda.get_device_name(0)}")
   print(f"GPU Count: {torch.cuda.device_count()}")
   print(f"CUDA Version: {torch.version.cuda}")
   # Enable cuDNN auto-tuner for better performance
   torch.backends.cudnn.benchmark = True
else:
   print("  Attention: CUDA n'est pas disponible. Entraînement lent sur CPU.")
   print("Installer CUDA-enabled PyTorch avec pip et la bonne version de cuda")

# Quelques variables globales
DATA_PATH = 'ptbxl/'
SAMPLING_RATE = 100  # Hz
BATCH_SIZE = 32  # à augmenter si la mémoire GPU le permet
NUM_EPOCHS = 50
LEARNING_RATE = 0.001
MIN_SAMPLES_PER_CLASS = 50  # Minimum d'ecg à avoir dans une classe donnée
```

1.4 Chargement des données

[ ]:
```
print("\n" + "="*100)
X, Y = load_ptbxl_data(DATA_PATH, SAMPLING_RATE)
```

1.5 Fonction de traitement des étiquettes

[ ]:
```
def preprocess_labels(Y, data_path='./ptbxl/', target_classes=['NORM', 'MI',␣
↪'STTC', 'CD', 'HYP'], min_samples=50):
    """
```

```
    Extration et traitement des etiquettes de diagnostic avec un filtrage␣
↪automatique
    Args:
        Y: dataframe des metatdata
        data_path: chemin vers le dossier PTB-XL (pour le fichier␣
↪scp_statements.csv)
        target_classes: liste des classes de diagnostic à prédire par le modèle␣
↪de classification
        min_samples: nombre minimal d'ecg par classe (pour considérer la classe)
    Returns:
        Binary labels for classification, filtered class names,␣
↪MultiLabelBinarizer
    """
    # Load SCP statements to get superclass mappings
    scp_statements_file = os.path.join(data_path, 'scp_statements.csv')
    if os.path.exists(scp_statements_file):
        scp_statements = pd.read_csv(scp_statements_file, index_col=0)
        scp_statements = scp_statements[scp_statements.diagnostic == 1]

        def aggregate_diagnostic(scp_codes):
            """Aggregation des codes SCP dans des classes de diagnostics"""
            labels = []
            for key in scp_codes.keys():
                if key in scp_statements.index:
                    superclass = scp_statements.loc[key].diagnostic_class
                    if superclass in target_classes:
                        labels.append(superclass)
            return list(set(labels))  # Remove duplicates

        Y['diagnostic_labels'] = Y.scp_codes.apply(aggregate_diagnostic)
    else:
        print(f"Warning: {scp_statements_file} not found. Using direct SCP code␣
↪matching.")
        def extract_labels(scp_codes, target_classes):
            labels = []
            for tc in target_classes:
                if tc in scp_codes:
                    labels.append(tc)
            return labels

        Y['diagnostic_labels'] = Y.scp_codes.apply(lambda x: extract_labels(x,␣
↪target_classes))

    # Convert to binary format
    mlb = MultiLabelBinarizer(classes=target_classes)
    y_binary = mlb.fit_transform(Y['diagnostic_labels'])
```

```
    print(f"\n=== Distribution initiales des classes ===")
    for i, label in enumerate(target_classes):
        count = y_binary[:, i].sum()
        percentage = (count / len(y_binary)) * 100
        print(f"{label}: {count} samples ({percentage:.2f}%)")

    # Filter out classes with too few samples
    valid_classes = []
    valid_indices = []

    for i, label in enumerate(target_classes):
        count = y_binary[:, i].sum()
        if count >= min_samples:
            valid_classes.append(label)
            valid_indices.append(i)
        else:
            print(f"  Warning: {label} has only {count} samples (<␣
↪{min_samples}). Excluding from training.")

    if len(valid_classes) == 0:
        raise ValueError(f"No classes have at least {min_samples} samples. Try␣
↪lowering min_samples parameter.")

    # Keep only valid classes
    y_binary_filtered = y_binary[:, valid_indices]

    print(f"\n=== Les classes retenues ({len(valid_classes)} classes) ===")
    for i, label in enumerate(valid_classes):
        count = y_binary_filtered[:, i].sum()
        percentage = (count / len(y_binary_filtered)) * 100
        print(f"{label}: {count} samples ({percentage:.2f}%)")

    return y_binary_filtered, valid_classes, mlb
```

1.6 Filtrage et traitement automatique des étiquettes (classes)

[ ]:
```
print("\n" + "="*100)
print("Traitement des étiquettes")
print("="*100)
initial_target_classes = ['NORM', 'MI', 'STTC', 'CD', 'HYP']
y_binary, target_classes, mlb = preprocess_labels(Y, DATA_PATH,␣
↪initial_target_classes, MIN_SAMPLES_PER_CLASS)

if len(target_classes) == 0:
   print("\n ERREUR: Pas de classes avec un nombre suffisant d'ECG!")
```

```
    print(f"Try lowering MIN_SAMPLES_PER_CLASS (currently␣
↪{MIN_SAMPLES_PER_CLASS})")
    exit(1)
```

1.7 Fonction pour stratification apprentissage-validation-test

[ ]:
```
def stratified_split(X, y, test_size=0.3, random_state=42):
    """
    partage stratifié pour des données multi-étiquettes
    """
    if STRATIFIED_AVAILABLE and y.shape[1] > 1:
        print("stratified split for multi-label data...")
        msss = MultilabelStratifiedShuffleSplit(n_splits=1,␣
↪test_size=test_size, random_state=random_state)

        for train_idx, test_idx in msss.split(X, y):
            X_train, X_test = X[train_idx], X[test_idx]
            y_train, y_test = y[train_idx], y[test_idx]

        return X_train, X_test, y_train, y_test
    else:
        print("regular stratified split...")
        stratify_labels = y.argmax(axis=1) if y.shape[1] > 1 else y.ravel()
        return train_test_split(X, y, test_size=test_size,␣
↪random_state=random_state, stratify=stratify_labels)
```

1.8 Création d'une partition avec stratification

[ ]:
```
print("\n" + "="*100)
print("Partition avec stratification")
print("="*60)
# First split: train vs temp (val+test)
X_train, X_temp, y_train, y_temp = stratified_split(X, y_binary, test_size=0.3,␣
↪random_state=42)

# Second split: val vs test
X_val, X_test, y_val, y_test = stratified_split(X_temp, y_temp, test_size=0.5,␣
↪random_state=42)

print(f"Ensemble d'apprentissage: {len(X_train)} données")
print(f"Ensemble de validation: {len(X_val)} données")
print(f"Ensemble de test: {len(X_test)} données")
```

1.9 Vérification des effectifs des classes dans les trois jeux données

```
[ ]: print("\n=== Effectifs des classes ===")
     for i, class_name in enumerate(target_classes):
        train_count = y_train[:, i].sum()
        val_count = y_val[:, i].sum()
        test_count = y_test[:, i].sum()
        print(f"{class_name}: Train={train_count}, Val={val_count},␣
      ↪Test={test_count}")
```

1.10 Partie 1. Standarisation des ECG

Écrire une fonction **normalize_ecg** qui prend en entrée les trois ensemble d'ecg **X_train**, **X_val** et **X_test** et qui les standarise et qui renvoie les trois jeux de données standardisés ainsi que le objet généré l'instanciation de la classe **StandardScaler()**. Pour celà, nous allons avoir besoin de :

- Instancier un objet **StandardScaler()**
- Regrouper dans **X_train_reshaped** tous les signaux par dérivation à l'aide de la méthode **.rechape()** du jeu de données **X_train**
- Appliquer la méthode **.fit** de l'objet **scaler** sur le jeu de données **X_train_reshaped** pour les calculer les moyennes et les écarts types par dérivation.
- Appeler la méthode **.transform** sur les jeux de données **X_train**, **X_val** et **X_test** et renvoyer les versions standardisées.

1.11 Une classe utilitaire pour créer des tenseurs PyTorch

```
[ ]: class ECGDataset(Dataset):
         def __init__(self, signals, labels):
             self.signals = torch.FloatTensor(signals)
             self.labels = torch.FloatTensor(labels)

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

         def __getitem__(self, idx):
             return self.signals[idx], self.labels[idx]
```

1.12 Création des trois itérateurs sur les jeux de données apprentissage, validation et test

```
[ ]: train_dataset = ECGDataset(X_train, y_train)
     val_dataset = ECGDataset(X_val, y_val)
     test_dataset = ECGDataset(X_test, y_test)

     train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
     val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE)
     test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE)
```

1.13 Fonction pour tracer un ECG

Comme pour la lecture du jeu de données, cette fonction permet de tracer les 12 leads d'un ECG avant ou après standardisation.

[ ]:
```
def plot_ecg_sample(signal, title="ECG Signal"):
    """
    Courbes des 12 dérivation d'un ECG
    """
    lead_names = ['I', 'II', 'III', 'aVR', 'aVL', 'aVF', 'V1', 'V2', 'V3',␣
 ↪'V4', 'V5', 'V6']

    fig, axes = plt.subplots(12, 1, figsize=(15, 12))
    fig.suptitle(title, fontsize=16)

    for i in range(12):
        axes[i].plot(signal[:, i], linewidth=0.5)
        axes[i].set_ylabel(lead_names[i])
        axes[i].grid(True, alpha=0.3)
        if i < 11:
            axes[i].set_xticks([])

    axes[-1].set_xlabel('temps (données)')
    plt.tight_layout()
    plt.show()

## pour faire un test
plot_ecg_sample(X_train[0], "Un exemple d'ECG - 12 leads")
```

1.14 Partie 2. Création d'un modèle CNN_LSTM :

Cette question traite la partie centrale du devoir qui consiste à déclarer un réseau de neurones avec des couches convolutives, une couche LSTM pour le traitement de l'aspect temporel ainsi qu'une couche dense pour la classification. Pour plus de détails sur le fonctionnement d'une couche LSTM (vous pouvez lire la partie **10.1** du livre d2l. Compléter la déclaration de la classe suivante.

1.14.1 Quelques directives

- Compléter uniquement les lignes précédée d'un ##.
- Les filtres de convolution ne sont pas en dimension 2 comme pour les images.
- Identifier l'équivalent d'un canal d'une image dans le cas d'un ECG.
- Faire attention à la fonction d'activation de la couche de sortie pour une classification multi-labels.

[ ]:
```
class CNN_LSTM(nn.Module):
    """
    Architecture de type CNN-LSTM
    """
    def __init__(self, num_classes=5, num_leads=12):
```

```
        super(CNN_LSTM, self).__init__()

        # Couches CNN pour l'extraction d'information (feature)
        self.conv1 = ## Une première couche convolutive avec le bon nombre de␣
↪canaux composée d'un nombre de filtres
                     ## (entre 50 et 80) avec une taille de filtre (entre 5 et␣
↪10) et un padding (entre 2 et 5)

        self.bn1 =   ## Une couche de batch normalisation de avec la dimension␣
↪appropriée

        self.pool1 = ## Une couche de max pooling d'une fenêtre de taille 2.

        self.conv2 = ## Une seconde couche convolutive en doublant le nombre de␣
↪filtres par rapport à la précédente.
                     ## Une taille de filtre inférieure à la précédente et un␣
↪padding de 2.

        self.bn2 =   ## Une couche de batch normalisation de avec la dimension␣
↪appropriée

        self.pool2 = ## Une couche de max pooling d'une fenêtre de taille 2.


        # Couche LSTM pour modéliser la dimension temporelle
        self.lstm =  ## Une couche LSTM bidirectionnelle  (nn.LSTM) avec un␣
↪nombre d'états cachés égal à la dimension de son entrée,
                     ## batch_first = True


        # Les couches de classification
        self.fc1 =   ## Une couche linéaire avec un nombre neurones égal au␣
↪nombre d'états cachés de la couche LSTM.
                     ## (Attention au nombre d'entrées)

        self.dropout = ## Une couche de dropout à 50%

        self.fc2 =  ## Une couche linéaire de sortie avec le bon nombre␣
↪d'unités


        self.relu = ## Une fonction d'activation ReLU.

    # déclaration de fonction pass-avant du réseau de neurones
    def forward(self, x):
        # x shape: (batch, time_steps, leads) en entrée.
```

```
        x =  ## Utiliser une permutation pour avoir les derivations en seconde␣
↪dimension.

        x = self.relu(self.bn1(self.conv1(x)))
        x = self.pool1(x)

        x = self.relu(self.bn2(self.conv2(x)))
        x = self.pool2(x)

        x = ## Utiliser une permutation pour remettre la dimension temporelle␣
↪en seconde position



        x, _ = self.lstm(x)

        x = ## Récupérer uniquement la dernière composante temporelle

        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return  ## choisir et appliquer à x la bonne fonction d'activation␣
↪appropriée à la sortie du réseau de neurones
```

1.15 Partie 3. Entraînement du modèle

Compléter les deux bouts de code manquants au début de la fonction suivante.

[ ]:
```
def train_model(model, train_loader, val_loader, num_epochs=50, lr=0.001,␣
↪device='cuda', model_name='best_ecg_model'):
    """
    Entraînement du modèle
    """
    criterion = ## Choisir la bonne fonction de perte pour le problème de␣
↪classification traité ici.
    optimizer = ## Utiliser un algorithme d'optimisation de type Adam
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min',␣
↪patience=5)

    train_losses = []
    val_losses = []
    best_val_loss = float('inf')

    for epoch in range(num_epochs):
        # Training
```

```
        model.train()
        train_loss = 0
        for signals, labels in train_loader:
            signals, labels = signals.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(signals)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        train_loss /= len(train_loader)
        train_losses.append(train_loss)

        # Validation
        model.eval()
        val_loss = 0
        with torch.no_grad():
            for signals, labels in val_loader:
                signals, labels = signals.to(device), labels.to(device)
                outputs = model(signals)
                loss = criterion(outputs, labels)
                val_loss += loss.item()

        val_loss /= len(val_loader)
        val_losses.append(val_loss)

        scheduler.step(val_loss)

        print(f"Cycle {epoch+1}/{num_epochs} - Train Loss: {train_loss:.4f},␣
↪Val Loss: {val_loss:.4f}")

        # Enregistrer le meilleur modèle
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            # S'assurer que le répertoire existe et enregistrer
            model_dir = os.path.join('results', 'models')
            os.makedirs(model_dir, exist_ok=True)
            model_path = os.path.join(model_dir, f"{model_name}.pth")
            torch.save(model.state_dict(), model_path)

    return train_losses, val_losses
```

1.16 Quelques fonctions utilitaires

Les fonctions suivantes permettent d'évaluer, tracer les résultats de test d'un modèle de classification des ecg.

[ ]:
```python
def evaluate_model(model, test_loader, target_classes, device='cuda'):
    """
     Évaluation des performances d'un modèle
    """
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for signals, labels in test_loader:
            signals = signals.to(device)
            outputs = model(signals)
            all_preds.append(outputs.cpu().numpy())
            all_labels.append(labels.numpy())

    all_preds = np.vstack(all_preds)
    all_labels = np.vstack(all_labels)

    # convertion des probas en prédictions binaires
    binary_preds = (all_preds > 0.5).astype(int)

    # Calculate metrics
    print("\n=== Rapport de classification ===")
    print(classification_report(all_labels, binary_preds,
                                target_names=target_classes,
                                zero_division=0))

    # Calcul des scores AUC-ROC pour chaque classe
    print("\n=== Scores AUC-ROC ===")
    for i, label in enumerate(target_classes):
        try:
            # Check if we have both positive and negative samples
            if len(np.unique(all_labels[:, i])) > 1:
                auc_score = roc_auc_score(all_labels[:, i], all_preds[:, i])
                print(f"{label}: {auc_score:.4f}")
            else:
                print(f"{label}: N/A (only one class present in test set)")
        except Exception as e:
            print(f"{label}: N/A (error: {str(e)})")

    return all_preds, all_labels, binary_preds
```

```python
def plot_training_history(train_losses, val_losses):
    """
     figure de l'historique d'entraînement
    """
    plt.figure(figsize=(10, 5))
    plt.plot(train_losses, label='Training Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training History')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

def plot_confusion_matrix(y_true, y_pred, class_names):
    """
    figure de matrices de confusion pour chaque classe
    """
    num_classes = len(class_names)
    rows = (num_classes + 2) // 3
    cols = min(3, num_classes)

    fig, axes = plt.subplots(rows, cols, figsize=(5*cols, 5*rows))
    if num_classes == 1:
        axes = [axes]
    else:
        axes = axes.ravel() if num_classes > 1 else [axes]

    for i, class_name in enumerate(class_names):
        cm = confusion_matrix(y_true[:, i], y_pred[:, i], labels=[0, 1])
        sns.heatmap(cm, annot=True, fmt='d', ax=axes[i], cmap='Blues')
        axes[i].set_title(f'{class_name}')
        axes[i].set_ylabel('Vraies')
        axes[i].set_xlabel('Predites')

    # cacher ce qui n'est pas utilisé
    for i in range(num_classes, len(axes)):
        axes[i].axis('off')

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

1.17 Partie 4. Entraînement et évaluation du modèle

1.17.1 Quelques directives

- Entraîner le modèle à l'aide de la fonction **train_model**
- Afficher l'historique d'entraînement à l'aide de la fonction **plot_training_history**

- Penser à sauvegarder le modèle à la fin de l'entraînement
- Charger les poids du modèle sauvegardé pour l'évaluer sur le jeu de données test à l'aide de la fonction **evaluate_model**
- Afficher une matrice de confusion par classe.
- Penser à utiliser l'instruction suivante, avant et après l'entraînement pour vider le cache de la GPU.

[ ]:
```
# avant et après l'entraînement
if torch.cuda.is_available():
   torch.cuda.empty_cache()
```

1.18 Instructions à respecter :

Le devoir peut être traité seul ou en binôme uniquement. La copie à rendre doit être générée en pdf uniquement à partir du notebook de travail et incluant les sorties (résultats de vos calcul). En absence des sorties des cellules de code, votre code ne sera pas testé donc insuffisant. Votre copie est à rendre sous la forme prenom_nom.pdf ou prenom1_nom1_prenom2_nom2.pdf. Vous pouvez me l'envoyer par mail ou déposer sur un drive si nécessaire. **Aucune copie fabriquée avec des bouts de captures d'écrans ne sera acceptée**.