Alzheimer’s Disease Detection Using EEG and fNIRS Signals

Team Members: Gangfeng Hu, Sangdae Nam, Niko Hams, Uli Prantz, Eric Ji, Matthew Zhou

Overview of Alzheimer’s Disease (AD)

Alzheimer’s Disease (AD) is a progressive neurological disorder characterized by memory loss, cognitive decline, and impaired daily functioning. It is classified into three stages:

- **NC (Normal Cognition):** Healthy individuals with no cognitive impairments.
- **MCI (Mild Cognitive Impairment):** An intermediate stage with noticeable cognitive changes but no significant impact on daily life.
- **AD (Alzheimer's Disease):** Severe cognitive decline affecting memory, reasoning, and independence.

Our goal is to classify individuals into these three categories based on EEG and fNIRS signals collected during cognitive tasks.

Introduction to EEG and fNIRS

- **Electroencephalography (EEG):** A non-invasive technique that measures electrical brain activity through electrodes on the scalp. It offers high temporal resolution, allowing the capture of rapid neural responses.
- **Functional Near-Infrared Spectroscopy (fNIRS):** A non-invasive method that monitors brain activity by measuring changes in oxygenated and deoxygenated hemoglobin. It provides insights into cerebral blood flow and oxygenation, complementing EEG's electrical activity data.

By combining EEG and fNIRS, we leverage their complementary strengths to enhance the accuracy of Alzheimer’s detection.

Cognitive Tasks

We utilize four tasks to collect EEG and fNIRS signals, each designed to probe different cognitive functions:

1. **Resting State:** Participants focus on a stationary white cross on a black screen for 60 seconds, providing a baseline measure of brain activity.
2. **Oddball Task:** Participants view alternating yellow (target) and blue (non-target) circles, pressing a button for yellow circles. This task measures attention and response inhibition.

3. **1-back Task:** Participants view a sequence of random numbers (1-3) and press a button if the current number matches the previous one. This task evaluates working memory and cognitive flexibility.
4. **Verbal Fluency Task:** Participants perform two types of language-related tasks:
   - **Phonemic Fluency:** Generate words starting with a specific letter.
   - **Semantic Fluency:** Generate words related to a given category (e.g., animals).
     This task assesses language processing and semantic memory.

By analyzing the EEG and fNIRS signals from these tasks, our project aims to classify participants into NC, MCI, or AD categories, facilitating early and accurate diagnosis of Alzheimer’s Disease.

In [1]:
```
!pip install -r ./requirements.txt --quiet
```

Methods

In this project, we replicated the machine learning methods described in the reference paper for Alzheimer’s Disease classification. Specifically, we implemented the following models:

1. **ExtraTrees Classifier:**
   A tree-based ensemble learning method that uses randomized node splitting and feature selection. It is computationally efficient, reduces variance to prevent overfitting, and measures feature importance effectively. This classifier was used in combination with Recursive Feature Elimination with Cross-Validation (RFECV) to identify the most discriminative features from EEG and fNIRS signals.

2. **Multi-Layer Perceptron (MLP):**
   A neural network-based model that captures complex non-linear relationships in the data. The MLP used fully connected layers and was trained with backpropagation to learn meaningful representations from the extracted features.

Both methods were evaluated using accuracy, F1 score, and AUC as performance metrics. By comparing these models, we aimed to assess their effectiveness in classifying participants into NC, MCI, and AD categories based on hybrid EEG-fNIRS features.

In [9]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import pandas as pd
import numpy as np
import time

from torch.utils.data import Dataset, DataLoader
```

```python
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
```

In [3]:
```python
file_path = "./csv_folder/Experiment1/RFECV-5secEEGPSD_FullFnirsPSD_FullFnirsTim
dataset = pd.read_csv(file_path)
```

Method 1: ExtraTrees Classifier with RFECV

In [5]:
```python
X = dataset.iloc[:,2:]
y = dataset.iloc[:,1]
```

In [ ]:
```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
```

In [7]:
```python
clf = ExtraTreesClassifier(n_estimators=100, random_state=42)
clf.fit(X_train, y_train)
```

Out[7]:
```
▾ ExtraTreesClassifier
ExtraTreesClassifier(random_state=42)
```

In [8]:
```python
y_pred = clf.predict(X_test)
print(clf.score(X_test, y_test))
print(classification_report(y_test, y_pred))
```

```
0.7931034482758621
              precision    recall  f1-score   support

           0       1.00      0.88      0.93         8
           1       0.73      0.85      0.79        13
           2       0.71      0.62      0.67         8

    accuracy                           0.79        29
   macro avg       0.82      0.78      0.80        29
weighted avg       0.80      0.79      0.79        29
```

Method 2: Multi-Layer Perceptron (MLP)

In [11]:
```python
class SignalDataset(Dataset):
    def __init__(self, dataframe):
        self.data = dataframe
        self.features = dataframe.iloc[:, 2:].values.astype(np.float32)  # Featu
        self.labels = dataframe['label'].values.astype(np.int64)  # Label column

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

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

In [12]:
```python
signal_dataset = SignalDataset(dataset)
```

In [13]:
```
train_size = int(0.8 * len(signal_dataset))
test_size = len(signal_dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(signal_dataset, [tra

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)
```

In [14]:
```
class MLPClassifier(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(MLPClassifier, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim)
        )

    def forward(self, x):
        return self.net(x)
```

In [15]:
```
model = MLPClassifier(input_dim=151, hidden_dim=32, output_dim=3)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.01)
```

In [16]:
```
def train_model(model, train_loader, criterion, optimizer, scheduler, epochs=20)
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for features, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(features)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        scheduler.step()

        # if (epoch + 1) % 5 == 0:
        #     print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss:.4f}, Learni
```

In [17]:
```
def test_model(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for features, labels in test_loader:
            outputs = model(features)
            predicted = torch.max(outputs, 1)[1]
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return 100 * correct / total
```

In [18]:
```
def find_best_model(train_loader, test_loader, input_dim, hidden_dim, output_dim
    best_acc = 0
    best_model = None
    for i in range(10):
        model = MLPClassifier(input_dim, hidden_dim, output_dim)
```

```
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(model.parameters(), lr=0.01)
        scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.0
        train_model(model, train_loader, criterion, optimizer, scheduler, epochs
        acc = test_model(model, test_loader)
        if acc > best_acc:
            best_acc = acc
            best_model = model
    return best_model, best_acc
```

In [19]:
```
best_model, best_acc = find_best_model(train_loader, test_loader, 151, 32, 3)

print(f"Best Accuracy: {best_acc:.2f}%")

cur_time = time.strftime("%Y%m%d-%H%M%S")
torch.save(best_model.state_dict(), f"./model/mlp_{cur_time}_acc{int(best_acc)}.
```

```
Best Accuracy: 86.21%
```

In [20]:
```
y_pred = best_model(torch.tensor(X_test.values).float()).argmax(dim=1)

print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.89      1.00      0.94         8
           1       0.90      0.69      0.78        13
           2       0.70      0.88      0.78         8

    accuracy                           0.83        29
   macro avg       0.83      0.86      0.83        29
weighted avg       0.84      0.83      0.83        29
```

In [ ]: