Source Code

```
from google.colab import drive

drive.mount('/content/drive/')

import pandas as pd

import zipfile

import os


# Path to your ZIP file

zip_path = "/content/EEG dataset of children with learning disabilities (LD).zip"


# Destination folder

extract_path = "/content/drive/MyDrive/learning disorder/EEG_dataset_extracted"


# Ensure the directory exists

os.makedirs(extract_path, exist_ok=True)


# Unzipping the file

with zipfile.ZipFile(zip_path, 'r') as zip_ref:

    zip_ref.extractall(extract_path)


print(f"✅ Files extracted to: {extract_path}")

df1=pd.read_csv("/content/drive/MyDrive/learning disorder/EEG_dataset_extracted/EEG dataset of children with learning disabilities (LD)/EEG dataset for children with learning disabilities/LD1_ec.csv")

import os
```

```
import pandas as pd

# Define the root directory
directory = "/content/drive/MyDrive/learning disorder/EEG_dataset_extracted"

# Initialize an empty list to store DataFrames
dataframes = []

# Walk through all subdirectories and find CSV files
for root, _, files in os.walk(directory):
    for file in files:
        if file.endswith(".csv"):  # Check if file is a CSV
            file_path = os.path.join(root, file)  # Full path to the file
            try:
                df = pd.read_csv(file_path)  # Read CSV file
                if not df.empty:
                    df["class"] = file  # Add filename as a new column
                    dataframes.append(df)
                else:
                    print(f"Skipping empty file: {file}")
            except Exception as e:
                print(f"Error reading {file}: {e}")

# Combine all DataFrames if there are valid files
if dataframes:
    combined_df = pd.concat(dataframes, ignore_index=True)
```

```
    print("Combined DataFrame created successfully!")
    print(combined_df.head())  # Show first few rows
else:
    print("No valid CSV data to combine.")

# Optionally, save the combined DataFrame to a CSV file
output_path = "/content/drive/MyDrive/learning disorder/combined_data.csv"
combined_df.to_csv(output_path, index=False)
print(f"Combined CSV saved to {output_path}")
from sklearn.preprocessing import LabelEncoder

# Initialize the LabelEncoder
label_encoder = LabelEncoder()

# Fit and transform the 'class' column
combined_df['class'] = label_encoder.fit_transform(combined_df['class'])

# Display unique class labels and their assigned numbers
class_mapping = dict(zip(label_encoder.classes_,
label_encoder.transform(label_encoder.classes_)))
print("Class Label Mapping:", class_mapping)

# Display first few rows
print(combined_df.head())

import torch
```

```
import torch.nn as nn
import torch.optim as optim
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, LabelEncoder
from torch.utils.data import DataLoader, TensorDataset

df = combined_df # Ensure your dataset is correctly formatted

# Handle NaN values (Option: Replace with Mean)
df.fillna(df.mean(), inplace=True)

# Separate features and target
X = df.iloc[:, :-1].values  # All columns except the last one
y = df.iloc[:, -1].values   # Last column (target class)

# Normalize features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Encode target labels
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

# Convert to PyTorch tensors
X_tensor = torch.tensor(X, dtype=torch.float32)
```

```
y_tensor = torch.tensor(y, dtype=torch.long)

# Reshape for RNN (batch_size, sequence_length, features)
X_tensor = X_tensor.view(X_tensor.shape[0], 1, X_tensor.shape[1])

# Create DataLoader
dataset = TensorDataset(X_tensor, y_tensor)
dataloader = DataLoader(dataset, batch_size=64, shuffle=True)

# Define RNN Model
class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNNModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)  # Initialize hidden
state
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])  # Take last time step's output
        return out

# Model parameters
```

```python
input_size = X.shape[1]
hidden_size = 64
num_layers = 2
num_classes = len(np.unique(y))
model = RNNModel(input_size, hidden_size, num_layers, num_classes)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train Model
epochs = 20
for epoch in range(epochs):
    total_loss = 0
    for inputs, labels in dataloader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f'Epoch [{epoch+1}/{epochs}], Loss: {total_loss/len(dataloader):.4f}')

# Save Model
torch.save(model.state_dict(), "rnn_model.pth")
print("Training Complete. Model saved.")
```

```
import torch
import torch.nn as nn
import torch.optim as optim
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, TensorDataset

# Load dataset (Ensure your dataset is correctly formatted)
df = combined_df  # Ensure this variable is properly defined

# Handle NaN values (Option: Replace with Mean)
df.fillna(df.mean(), inplace=True)

# Separate features and target
X = df.iloc[:, :-1].values  # All columns except the last one
y = df.iloc[:, -1].values   # Last column (target class)

# Normalize features
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

```
# Encode target labels
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32).view(X_train.shape[0], 1,
X_train.shape[1])
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32).view(X_test.shape[0], 1,
X_test.shape[1])
y_test_tensor = torch.tensor(y_test, dtype=torch.long)

# Create DataLoaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# Define RNN Model
class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNNModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
```

```
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)

        self.fc = nn.Linear(hidden_size, num_classes)


    def forward(self, x):

        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)  # Initialize hidden
state

        out, _ = self.rnn(x, h0)

        out = self.fc(out[:, -1, :])  # Take last time step's output

        return out


# Device setup (Use GPU if available)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


# Model parameters

input_size = X.shape[1]

hidden_size = 64

num_layers = 2

num_classes = len(np.unique(y))


model = RNNModel(input_size, hidden_size, num_layers, num_classes).to(device)


# Loss and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)


# Train Model
```

```
epochs = 20
for epoch in range(epochs):
    total_loss = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f'Epoch [{epoch+1}/{epochs}], Loss: {total_loss/len(train_loader):.4f}')

# Save Model
torch.save(model.state_dict(), "rnn_model.pth")
print("Training Complete. Model saved.")

# Ensure model is in evaluation mode
model.eval()

# Initialize lists for predictions and actual labels
all_preds = []
all_labels = []

# Iterate over the test data
with torch.no_grad():
```

```
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)  # Get the predicted class

        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# Compute confusion matrix
cm = confusion_matrix(all_labels, all_preds)

# Get class names (Ensure they are strings)
class_names = [str(cls) for cls in label_encoder.classes_]

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_names,
yticklabels=class_names)
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()

# Print classification report
print("Classification Report:\n")
print(classification_report(all_labels, all_preds, target_names=class_names))
```

```
import torch
import torch.nn as nn
import torch.optim as optim
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, TensorDataset

# Load dataset (Ensure your dataset is correctly defined)
df = combined_df  # Ensure this variable is properly assigned

# Handle NaN values (Option: Replace with Mean)
df.fillna(df.mean(), inplace=True)

# Separate features and target
X = df.iloc[:, :-1].values  # All columns except the last one
y = df.iloc[:, -1].values   # Last column (target class)

# Normalize features
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

```
# Encode target labels
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

# Split data into training and test sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)

# Reshape for RNN/LSTM (batch_size, sequence_length, features)
X_train_tensor = X_train_tensor.view(X_train_tensor.shape[0], 1, X_train_tensor.shape[1])
X_test_tensor = X_test_tensor.view(X_test_tensor.shape[0], 1, X_test_tensor.shape[1])

# Create DataLoaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)  # Defined test_loader

# Define LSTM Model
class LSTMModel(nn.Module):
```

```python
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)  # LSTM needs cell
state
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])  # Take last time step's output
        return out

# Device setup (Use GPU if available)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Model parameters
input_size = X.shape[1]
hidden_size = 64
num_layers = 2
num_classes = len(np.unique(y))

model = LSTMModel(input_size, hidden_size, num_layers, num_classes).to(device)
```

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

# Train Model
epochs = 20
for epoch in range(epochs):
    total_loss = 0
    model.train()  # Set model to training mode
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f'Epoch [{epoch+1}/{epochs}], Loss: {total_loss/len(train_loader):.4f}')

# Save Model
torch.save(model.state_dict(), "lstm_model.pth")
print("Training Complete. Model saved.")

# Evaluation Mode
model.eval()
```

```
# Initialize lists for predictions and actual labels
all_preds = []
all_labels = []

# Iterate over the test data
with torch.no_grad():
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)  # Get the predicted class

        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# Compute confusion matrix
cm = confusion_matrix(all_labels, all_preds)

# Get class names (Ensure they are strings)
class_names = [str(cls) for cls in label_encoder.classes_]

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_names,
yticklabels=class_names)
plt.xlabel("Predicted")
plt.ylabel("Actual")
```

```
plt.title("Confusion Matrix")
plt.show()

# Print classification report
print("Classification Report:\n")
print(classification_report(all_labels, all_preds, target_names=class_names))
pip install gradio
X.shape
df
torch.save(model.state_dict(), "lstm_model.pth")
print("Training Complete. Model saved.")
import torch
import joblib
import numpy as np
from sklearn.preprocessing import StandardScaler, LabelEncoder

# Save the model
torch.save(model.state_dict(), "lstm_model.pth")

 # Ensure X_train is from your training dataset
joblib.dump(scaler, "scaler.pkl")

# Save the label encoder
 # Ensure y_train is from your training dataset
joblib.dump(label_encoder, "label_encoder.pkl")
```

```python
print("Model, Scaler, and Label Encoder saved.")
import gradio as gr
import torch
import joblib
import numpy as np

# Load the model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Modify the input_size and num_classes to match the saved model
input_size = 19  # Update this to 19, as the saved model expects 19 input features
hidden_size = 64
num_layers = 2
num_classes = 7  # Update this to 7, as the saved model expects 7 output classes

# Define the model class (Ensure it matches the architecture of the saved model)
class LSTMModel(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = torch.nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = torch.nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
```

```
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)  # LSTM needs cell
state
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])  # Take last time step's output
        return out

# Initialize the model
model = LSTMModel(input_size=input_size, hidden_size=hidden_size, num_layers=num_layers,
num_classes=num_classes)
model.load_state_dict(torch.load("lstm_model.pth"))
model.to(device)
model.eval()

# Load the scaler and label encoder
scaler = joblib.load("scaler.pkl")
label_encoder = joblib.load("label_encoder.pkl")

# Define the prediction function
def predict(*inputs):
    # Convert input into numpy array and reshape
    inputs = np.array(inputs).reshape(1, -1)

    # Normalize the input
    inputs = scaler.transform(inputs)

    # Convert input into tensor for prediction
    inputs_tensor = torch.tensor(inputs, dtype=torch.float32).view(1, 1, -1).to(device)
```

```
    # Get model output
    with torch.no_grad():
        output = model(inputs_tensor)
        _, predicted = torch.max(output, 1)

    # Map the predicted class based on the indices you provided
    pred_class_idx = predicted.item()


    if pred_class_idx in [0, 1, 3, 4]:
        predicted_label = "ADHD and Dyslexia"
    elif pred_class_idx in [5, 6]:
        predicted_label = "ASD"
    elif pred_class_idx == 2:
        predicted_label = "Dyslexia and ASD"
    else:
        predicted_label = "Unknown"

    return predicted_label

# Create Gradio interface
inputs = [gr.Number(label=f"Channel {i}") for i in range(1, 20)]  # 19 channels (adjust as per your data)
outputs = gr.Textbox(label="Predicted Class")


# Add a submit button
```

```
interface = gr.Interface(
    fn=predict,
    inputs=inputs,
    outputs=outputs,
    live=False,  # Ensure prediction only happens after submit
    allow_flagging="never",  # Optional: to disable flagging if you want
    title="EEG Channel Prediction",  # Optional: title for the UI
    description="Enter the EEG channel data, then click 'Submit' to get the predicted class."  # Optional: description
)

# Launch the interface
interface.launch(debug=True)
```