BE 521: Homework 8 Questions

Deep Learning

Spring 2025

Developed by William Ojemann

64 points

Due: March 27th, 2025

Objective: Apply deep learning to a brain-computer interfacing problem

AI Usage Notice

The use of artificial intelligence tools (e.g., large language models, code assistants) is permitted. However, students must explicitly state the specific ways AI was used in completing their work. Failure to disclose AI usage may result in an oral examination to assess understanding, at the discretion of Dr. Litt.

If AI was used in the completion of this assignment, please provide a statement below:

[Enter your statement here]

Overview

The goal of this homework is to add time series prediction and deep learning to your brain-computer interfacing toolkit. In this homework we will be developing a deep learning framework that we can use for seizure detection loosely following the procedure described by Bhattacharya et al., 2023.

If you remember from the reading... The model they describe is first trained in autoregression, and then they use the error of the model to determine whether the current sample is a seizure clip or not.

The first section will guide you through how to build a Long-Short Term Memory (`LSTM`) neural network using a toy signal to help ensure that the model is working. In the second section, we will apply the model we built to detecting seizures from one patient: `HUP235` .

In [ ]:
```
!jupyter nbconvert --to html Prakriti_HW7.ipynb
```

In [119…
```
# iEEG imports
!pip install git+https://github.com/ieeg-portal/ieegpy.git # Install ieegpy toolbox directly from github
from ieeg.auth import Session

# Imports for deep learning
import random
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

# Scientific computing imports
import numpy as np
import scipy as sc
import pandas as pd
from scipy.linalg import hankel
from tqdm import tqdm
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score

# Plotting
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

```
Collecting git+https://github.com/ieeg-portal/ieegpy.git
  Cloning https://github.com/ieeg-portal/ieegpy.git to /tmp/pip-req-build-nfks9dsi
  Running command git clone --filter=blob:none --quiet https://github.com/ieeg-portal/ieegpy.git /tmp/pip-req-build-nfks9dsi
  Resolved https://github.com/ieeg-portal/ieegpy.git to commit 080bfa42a8503380ef164b5e7b116613f75073bb
  Preparing metadata (setup.py) ... done
Requirement already satisfied: deprecation in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.1.0)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.32.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.0.2)
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.2.2)
Requirement already satisfied: pennprov==2.2.4 in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.2.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from pennprov==2.2.4->ieeg==1.6) (2025.1.31)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.11/dist-packages (from pennprov==2.2.4->ieeg==1.6) (2.8.2)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.11/dist-packages (from pennprov==2.2.4->ieeg==1.6) (1.17.0)
Requirement already satisfied: urllib3>=1.23 in /usr/local/lib/python3.11/dist-packages (from pennprov==2.2.4->ieeg==1.6) (2.3.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from deprecation->ieeg==1.6) (24.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas->ieeg==1.6) (2025.1)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas->ieeg==1.6) (2025.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests->ieeg==1.6) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests->ieeg==1.6) (3.10)
```

In [119…

```python
def set_seed(seed):
  np.random.seed(seed)
  torch.manual_seed(seed)
  random.seed(seed)
set_seed(5210)
```

0. Introduction (2pts)

Before we get started with coding, describe how the model in Bhattacharya et al. works. What difference between interictal (time between seizures that is considered to be "normal" EEG) and ictal (Seizure activity that occurs in patients with epilepsy) data are they leveraging to build a seizure detector? (2 pts)

The model in Bhattacharya et al. (2023) uses an unsupervised approach to detect seizures in EEG by framing it as an anomaly detection problem. It employs a Long Short-Term Memory (LSTM) network trained on a one-second interictal EEG segment to predict the next data point in a time series (autoregression). Specifically, for each 12-sample window, the LSTM predicts the 13th sample, and the mean squared error (MSE) between the predicted and actual values is calculated. These MSE values (prediction losses) are then clustered using a density-based algorithm to classify segments as interictal (low loss) or ictal (high loss). The key difference they leverage is that interictal EEG data exhibit more predictable, regular patterns, resulting in lower prediction errors, while ictal EEG data, due to their chaotic and abnormal nature (e.g., spike-wave discharges), produce higher and more variable prediction errors, making them detectable as anomalies.

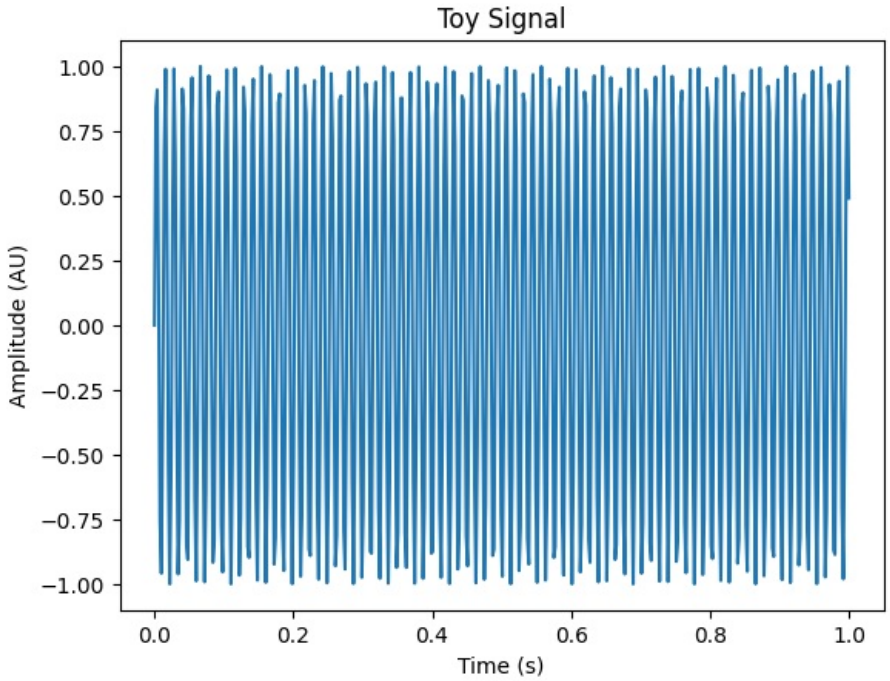
1. LSTMs, autoregression, and deep learning

Because this homework is more about deep learning, we'll provide some of the basic data for you. The following cell will create a sinusoidal signal with frequency $\frac{fs}{2\pi}$, for the duration of data_len (in seconds), and n_channels number of channels.

In [119…

```python
fs = 500
data_len = 1 # seconds
n_channels = 1
data = np.sin(torch.arange(0,data_len*fs,dtype=torch.float64).expand(n_channels,data_len*fs).T)
# data = torch.arange(0,data_len*fs,dtype=torch.float64).expand(n_channels,data_len*fs).T
sns.lineplot(x=np.linspace(0,1,fs),y=data.squeeze())
plt.ylabel('Amplitude (AU)');
plt.xlabel('Time (s)');
plt.title('Toy Signal');
```

```
<ipython-input-1193-cc3bc51974da>:4: DeprecationWarning: __array_wrap__ must accept context and return_scalar arguments (positionally) in the future. (Deprecated NumPy 2.0)
  data = np.sin(torch.arange(0,data_len*fs,dtype=torch.float64).expand(n_channels,data_len*fs).T)
```



If you remember from Bhattacharya et al., in order to detect seizures they split 1 second clips into 13 sample sequences and then used the first 12 samples as an input to predict the 13th. This can be generalized to: input samples $X = [x_i, x_{i+1}, \ldots, x_{i+11}]$ to predict y = x_{i+12}. This type of prediction, taking in samples from a previous time point to predict the next time point is called autoregression.

To facilitate an autoregression problem, we need to turn our 1 channel x 500 samples signal into a matrix where each row is a seperate input/prediction pair, or "sequence". More specifically, we will be transforming the one-second clip into a matrix of i sequences by j = 13 (12 input + 1 output) samples using something called a hankel matrix `scipy.linalg.hankel`.

To build a Hankel matrix, you can also think of each sequence as a window, so each row in the matrix should be a unique window, or sequence, that we're going to use to train our auto regression model to take in the 12 previous time points and predict the 13th. Because we have 500 samples in total, and require 13 continuous points to build a window, we can only have 500 - 13 + 1 = 488 windows/sequences.

In the example shown below, they have 4 input samples, 1 output sample, and 1 channel on a signal with 9 samples in it. So the dimensions of the Hankel matrix are *9 - (4+1) + 1 = 5 sequences* x *(4 + 1) samples in a sequence* x *1 channel*.

$$\begin{pmatrix} h_1 & h_2 & h_3 & h_4 & h_5 \\ h_2 & h_3 & h_4 & h_5 & h_6 \\ h_3 & h_4 & h_5 & h_6 & h_7 \\ h_4 & h_5 & h_6 & h_7 & h_8 \\ h_5 & h_6 & h_7 & h_8 & h_9 \end{pmatrix}$$

1 (4 pts)

1a

Create a tensor called `data_mat` with the following dimensions: i = 488 sequences, j = 13 (input + output) samples, and k = 1 channels. Print out the dimensions of your hankel matrix (2 pts).

Now, becuase we're going to be using the deep learning package PyTorch, we want our `data_mat` to be a `torch.tensor` rather than an `np.array`. Cast the `data_mat` to a `torch.tensor` (1 pt)

Hint: assuming there's just one channel and it is one second of data sampled at 500Hz, given m and n the first row should be $[x_1, x_2, x_3, \ldots, x_{13}]$ and the last row should be $[x_{488}, x_{489}, x_{490}, \ldots, x_{500}]$. The first column should then be all of the values from x_1 to x_{488}

Hint: you can also think of this as 13 sample, left aligned windows with a stride of 1 sample

Hint: you can use the commented out torch.arange signal to debug your hankel matrix

In [119…
```python
# Create your hankel matrix here...

from scipy.linalg import hankel

fs = 500
data_len = 1  # seconds
n_channels = 1
data = np.sin(torch.arange(0, data_len * fs, dtype=torch.float64).expand(n_channels, data_len * fs).T).numpy()

# Create Hankel matrix: 488 sequences × 13 samples
data_mat = hankel(data[:488], data[487:500])  # First 488 as column, last 13 as row
data_mat = data_mat[:, :, np.newaxis]  # Add channel dimension: 488 × 13 × 1

# Print dimensions
print("Hankel matrix dimensions:", data_mat.shape)

# Cast to torch.tensor
data_mat = torch.tensor(data_mat, dtype=torch.float32)
```

```
Hankel matrix dimensions: (488, 13, 1)
```

```
<ipython-input-1194-0d79a1ecf4d6>:8: DeprecationWarning: __array_wrap__ must accept context and return_scalar ar
guments (positionally) in the future. (Deprecated NumPy 2.0)
  data = np.sin(torch.arange(0, data_len * fs, dtype=torch.float64).expand(n_channels, data_len * fs).T).numpy()
```

1b

Now we want to split up our `data_mat` into the inputs = 12 and outputs = 1 contained in each sequence. Create two `torch.tensor`s: `input_data` with num sequences x sequence inputs x 1 channel, and `target_data` with the num sequences x sequence output (1 pts).

Hint: For pytorch, it's important to keep the dimensionality of our inputs so that they have this format - num sequences x sequence length x input channels

In [119…
```python
# Prepare input and target data for the model here...
# Please cast your torch.tensors to type float to avoid downstream issues:
# my_tensor = my_tensor.float()

# Split into input (first 12 samples) and target (13th sample)
input_data = data_mat[:, :12, :]  # 488 × 12 × 1
target_data = data_mat[:, 12, :]  # 488 × 1

# Ensure float type
input_data = input_data.float()
target_data = target_data.float()

print("Input data shape:", input_data.shape)
print("Target data shape:", target_data.shape)
```

```
Input data shape: torch.Size([488, 12, 1])
Target data shape: torch.Size([488, 1])
```

2 (8 pts)

In this section we will be setting up our LSTM. We talked about different neural network architecutres in the last recitation, but we'll briefly recap what an LSTM is here. An LSTM, or Long Short-Term Memory model, is a type of recurrent neural network designed to process sequential data - like language or, in our case, EEG recordings. An LSTM traverses across a sequence, at each point updating the values in its hidden state based on the previous hidden state and the current element of the sequence. For a more in-depth explanation of how LSTMs "forget" irrelevant information or store hidden states, I recommend interacting with ChatGPT to guide your learning.

For this homework, we're going to just be applying PyTorch's implementation of LSTMs rather than diving into the theory.

2a

Like we talked about in the recitation, we can use a pytorch dataloader to help facilitate training and generating predictions using a pytorch model. Put your `input_data` and `target_data` into a TensorDataset, and then create a DataLoader from that dataset that outputs batches of 100 samples. Be sure to leave all other settings as defaults (1 pt).

We've already imported the classes `TensorDataset` and `DataLoader` for you to use in this notebook

In [119…
```python
# Create DataLoader for training

dataset = TensorDataset(input_data, target_data)
dataloader = DataLoader(dataset, batch_size=100, shuffle=False)
```

2b

Here, you're going to define the architecture of the LSTM model from the paper. We talked about coding neural networks in pytorch in

recitation, but for this question we're going to leave it up to you. We highly recommend you use ChatGPT and ask it something along the lines of "what are the components of a basic LSTM model in pytorch with..."

Create an LSTM architecture called `LSTMModel` that takes in 3 inputs when you initialize it: input_size, hidden_size, and output_size (4 pts).

Hint: Some quick definitions - the input size is the number of channels that the data has, in our case this is one. The hidden size is the size of the hidden state variable, and the output size is the number of channels that our data has, in our case this is one again. When defining the LSTM, you don't have to include the sequence length, since the model will just traverse across the whole sequence regardless of length.

Hint: While you're building this architecture, be sure to go line by line and ask chatGPT to explain any parts of the model code that are confusing. It can be a super useful debugging tool. You can also use https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html as a helpful resource

Initialize the model with the same input, hidden, and output size as the paper and print the model: `print(model)` . (2 pts)

It should look like this:

```
LSTMModel(
  (lstm): LSTM(1, 100, batch_first=True)
  (fc): Linear(in_features=100, out_features=1, bias=True)
)
```

In [119…
```python
# Define LSTM model
# Define LSTM model
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # x shape: (batch_size, seq_len, input_size)
        batch_size = x.size(0)
        h0 = torch.zeros(1, batch_size, self.hidden_size).to(x.device)
        c0 = torch.zeros(1, batch_size, self.hidden_size).to(x.device)
        out, _ = self.lstm(x, (h0, c0))  # out: (batch_size, seq_len, hidden_size)
        out = self.fc(out[:, -1, :])  # Take last time step: (batch_size, output_size)
        return out

# Initialize model
model = LSTMModel(input_size=1, hidden_size=100, output_size=1)
print(model)
```

```
LSTMModel(
  (lstm): LSTM(1, 100, batch_first=True)
  (fc): Linear(in_features=100, out_features=1, bias=True)
)
```

2c

Before we train the model, we need to define the `criterion` (the metric that we will use to evaluate the model's predictions, often called the loss function) and the `optimizer` (the algorithm that updates the weights of the model as it trains).

We use MSE - mean squared error - to evaluate the loss of auto regression because it allows us to effectivily compare the distance from the true value, y_i, and the value we predict, $\hat{y}_i$, using the following equation:

$$MSE = \frac{1}{N}\sum(y_i - \hat{y}_i)^2$$

Models that perform really poorly will predict values for $\hat{y}_i$ that are different from/unrelated to the value of y_i and will have a higher MSE as a result.

In accordance with the paper, create instances of an Adam `optimizer` object with a learning rate of 0.001, and an MSELoss `criterion` object (1 pts). This is another problem where ChatGPT can help navigate the material and code.

In [119…
```python
# Define loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

3 (6 pts)

Let's put it all together!

Create a training loop that trains your model for 100 epochs (iterations through the whole training dataset) using your dataloader (to load batches of data), model (to generate predictions), criterion or loss function (to evaluate the predictions against the targets), and optimizer (to update the model weights).

In layman's terms, your training loop is going to:

1. loop through each epoch, and within each epoch, loop through each batch in your dataloader
2. generate predictions for the 13th sample based on the previous 12 samples in each segment in your batch
3. calculate the loss comparing the true and predicted 13th samples across the whole batch
4. update your model's weights using the optimizer

ChatGPT will once again be your friend here for what to include in the loop as well as understanding what each step does.

The training loop should print out the loss*10000 in the last batch every 10 epochs (4 pts).

Hint: Your final loss at epoch 100 should be 0.0074. If it's close but not exact, try pressing "restart and run all" under the runtime tab. Make sure that you are not iterating through the dataloader in any cells before you train the model, this can mess up the random seed!

In [119…
```
# Training loop
# Training loop
num_epochs = 100
for epoch in range(num_epochs):
    for batch_input, batch_target in dataloader:
        optimizer.zero_grad()  # Clear gradients
        output = model(batch_input)  # Forward pass
        loss = criterion(output, batch_target)  # Compute loss
        loss.backward()  # Backward pass
        optimizer.step()  # Update weights

    if epoch % 10 == 9:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item()*10000:.4f}')
```

```
Epoch [10/100], Loss: 144.3850
Epoch [20/100], Loss: 0.6563
Epoch [30/100], Loss: 0.0158
Epoch [40/100], Loss: 0.0103
Epoch [50/100], Loss: 0.0098
Epoch [60/100], Loss: 0.0093
Epoch [70/100], Loss: 0.0089
Epoch [80/100], Loss: 0.0084
Epoch [90/100], Loss: 0.0079
Epoch [100/100], Loss: 0.0074
```

4 (8 pts)

4a

Now that we have a trained model, we need to find a way to generate the data needed to make predictions. Because our model will ultimately use the squared error loss of autoregression to make predictions about whether a clip is ictal or not, we need a prediction function that can take in our model and a new one-second clip and return the corresponding np.array of MSE values.

The function should take in the trained model and the new 1 second clip, and return an array of mse values for each sequence (12 sample input, 1 sample output) in the clip. (4pts)

Hint: When we're training the model, we can take the mean squared error across the entire batch using the criterion function. However, in this case we don't want the mean square error, we want the square error for each prediction: instead of $MSE = \frac{1}{N}\sum(y_i - \hat{y}_i)^2$*, we want* $SE = (y_i - \hat{y}_i)^2$*. This can be achieved by setting the batch size of your dataloader to one and using the same criterion function, or you can also write your own square error function and concatenate all of the squared error values across each batch.*

In [120…
```
def compute_mse(model, clip):
    model.eval()
    clip = torch.tensor(clip, dtype=torch.float32).numpy() if isinstance(clip, torch.Tensor) else clip
    hankel_mat = hankel(clip[:488], clip[487:500])[:, :, np.newaxis]  # 488 × 13 × 1
    hankel_tensor = torch.tensor(hankel_mat, dtype=torch.float32)
    input_data = hankel_tensor[:, :12, :]  # 488 × 12 × 1
    target_data = hankel_tensor[:, 12, :]  # 488 × 1

    dataset = TensorDataset(input_data, target_data)
    dataloader = DataLoader(dataset, batch_size=1, shuffle=False)

    mse_values = []
    with torch.no_grad():
        for batch_input, batch_target in dataloader:
            pred = model(batch_input)
            se = (pred - batch_target) ** 2  # Squared error per prediction
            mse_values.append(se.item())
```

```
    return np.array(mse_values)
```

4b

Let's build a function that we can use to visualize the autoregressive predictions the model is making. This won't be used to make ictal/interictal classifications, but is useful for understanding what the model is actually doing.

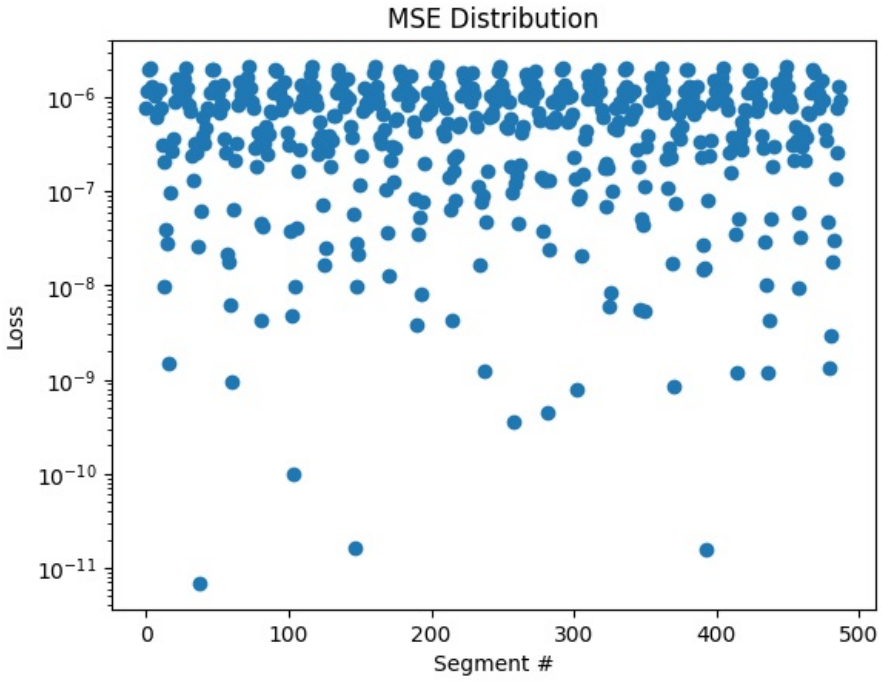
This function should take in the model and the clip and return an array of predictions for each segment in the clip. This should look very similar to your previous function. (2 pts)

In [120…
```
def predict_values(model, clip):
    model.eval()
    clip = torch.tensor(clip, dtype=torch.float32).numpy() if isinstance(clip, torch.Tensor) else clip
    hankel_mat = hankel(clip[:488], clip[487:500])[:, :, np.newaxis]
    hankel_tensor = torch.tensor(hankel_mat, dtype=torch.float32)
    input_data = hankel_tensor[:, :12, :]

    dataset = TensorDataset(input_data, torch.zeros_like(input_data))  # Dummy targets
    dataloader = DataLoader(dataset, batch_size=1, shuffle=False)

    predictions = []
    with torch.no_grad():
        for batch_input, _ in dataloader:
            pred = model(batch_input)
            predictions.append(pred.item())

    return np.array(predictions)
```

4c

Use your function to generate mse values for the data you used to train the model. Create a scatter plot of the mse distribution with segment # on the x axis and loss on the y axis. The plot should have a log-scale y axis. (1 pts)

In [120…
```
import matplotlib.pyplot as plt

mse_values = compute_mse(model, data)
plt.scatter(range(len(mse_values)), mse_values)
plt.yscale('log')
plt.xlabel('Segment #')
plt.ylabel('Loss')
plt.title('MSE Distribution')
plt.show()
```

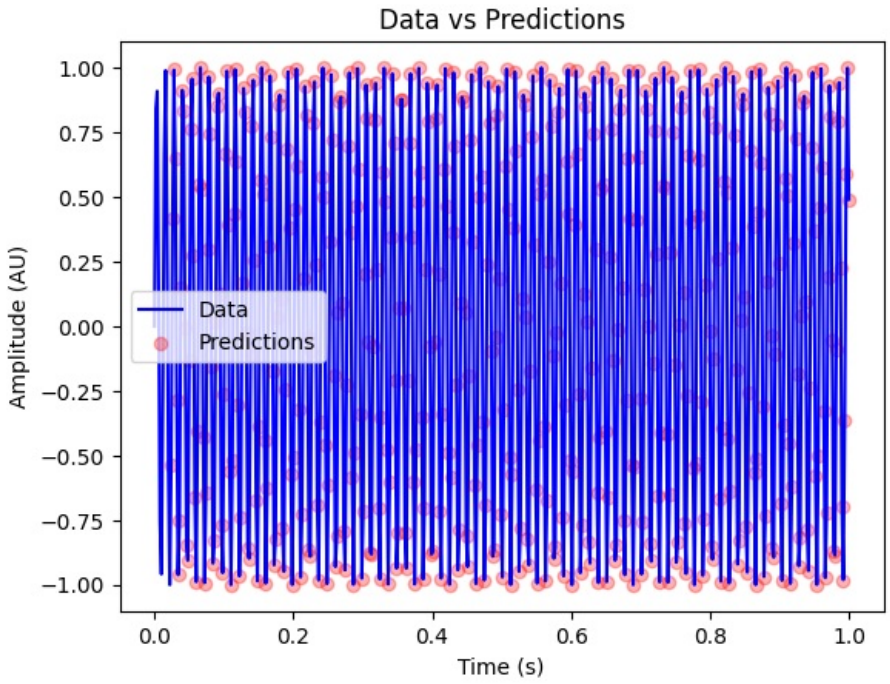


4d

Use your predict values function to generate predictions for each sample from x_{13} to x_{500}. plot the data values that you're generating predictions for as a blue lineplot, and then overlay a scatterplot of the predictions you're generating in red. Set the transparency of your red scatter plot to 0.3 using the "alpha" argument in plt.scatter() (1 pts).

In [120…
```
predictions = predict_values(model, data)
plt.plot(np.linspace(0, 1, fs), data, 'b-', label='Data')
plt.scatter(np.linspace(12/fs, 1, 488), predictions, color='red', alpha=0.3, label='Predictions')
```

```
plt.xlabel('Time (s)')
plt.ylabel('Amplitude (AU)')
plt.legend()
plt.title('Data vs Predictions')
plt.show()
```



2. Seizure detection

In [120…
```
# Resetting the seed for the next section!
set_seed(5210)
```

1 (5 pts)

`HUP235` is a patient with epilepsy who was implanted with intracranial EEG devices to help identify where in the brain their seizures were coming from. Your goal as a BCI/Deep Learning practitioner is to develop a seizure detection algorithm for their seizure onset channel - `LC01` - so that they can alert their caregiver when they've had a seizure.

1a

As part of the homework folder on canvas you should see `HUP235_seizure_times.csv`, which contains times for 11 seizures captured from HUP235. Load the CSV into the notebook as a `pd.DataFrame`. The csv should contain the following fields: "Patient": patient name, "iEEG_ID": id for iEEG.org, "Onset": seizure onset time, "Offset": seizure offset time, "Interictal": start time for a matched interictal clip. (1 pt)

In [120…
```
import pandas as pd

seizure_times = pd.read_csv('HUP235_seizure_times.csv')
print(seizure_times.head())
```

```
  Patient         iEEG_ID        Onset     Offset  Interictal
0  HUP235  HUP235_phaseII  107374.2078  107391.19  78574.8000
1  HUP235  HUP235_phaseII  107486.3563  107502.68  78686.3563
2  HUP235  HUP235_phaseII  107754.7900  107775.34  78954.7900
3  HUP235  HUP235_phaseII  116324.6629  116344.79  87524.6629
4  HUP235  HUP235_phaseII  116938.2674  116955.89  88135.5000
```

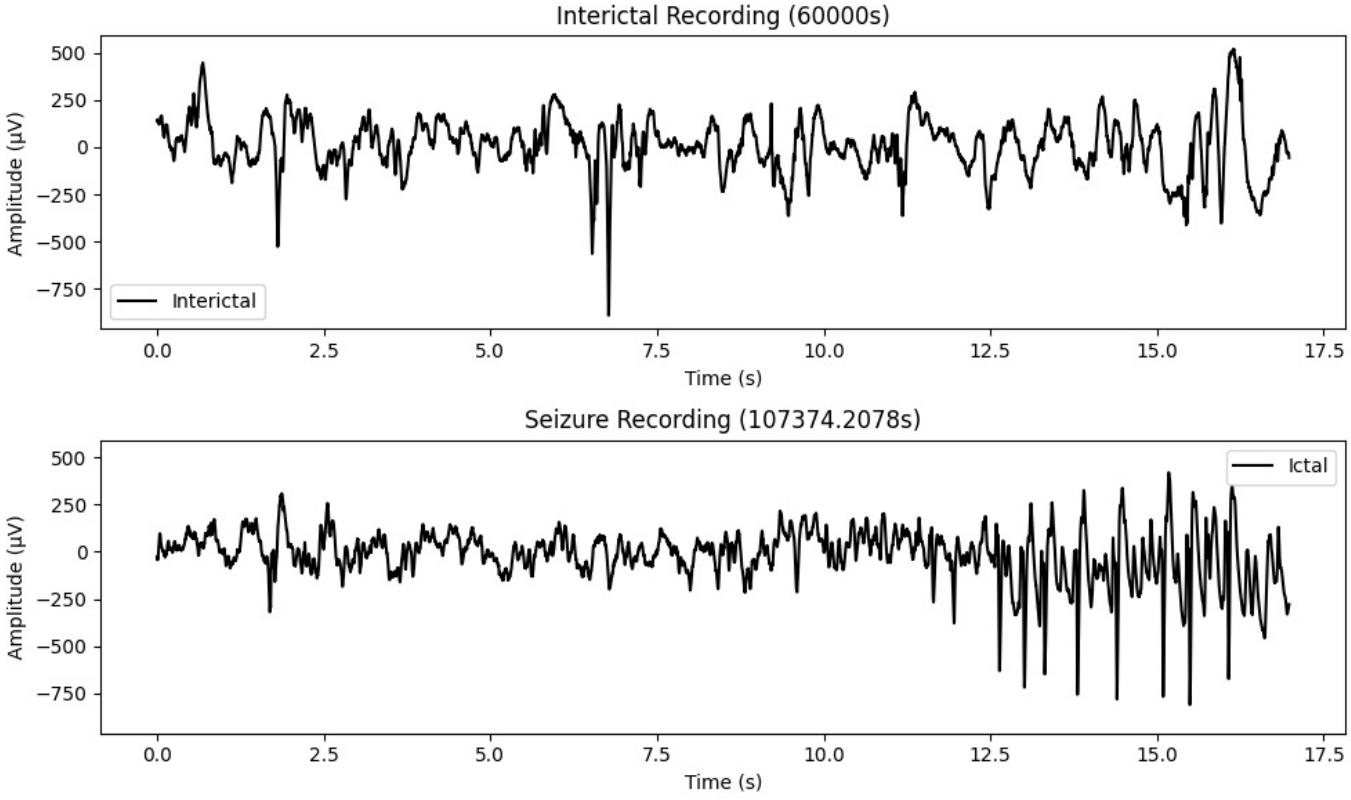
1b

We're first going to visualize the first seizure in the csv that has an onset at 107374.2078s. Generate a plot with two subplots: The first should show `LC01` of an interictal recording that starts at 60000 seconds and has the same duration as the seizure and the second should show channel `LC01` of the seizure recording. All plots should be black traces and have the same y axes. Be sure to label all axes and title the plots. (3 pts)

In [120…
```
from ieeg.auth import Session
import numpy as np
import matplotlib.pyplot as plt

# Load credentials and initialize session
with open('/content/pra_ieeglogin(9).bin', 'r') as f:
    session = Session('prasadpr', f.read().strip())
```

```python
# Open dataset
dataset = session.open_dataset("HUP235_phaseII")

# Get channel index for LC01
# Seizure times from CSV
seizure_onset = 107374.2078  # seconds
seizure_offset = 107391.19   # seconds
duration = seizure_offset - seizure_onset  # 16.9822s

# Interictal clip starts at 60000s
interictal_start = 60000
interictal_end = interictal_start + duration  # Not used directly, just for reference

# Convert to microseconds
seizure_start_us = int(seizure_onset * 1e6)
duration_us = int(duration * 1e6)  # Duration in microseconds
interictal_start_us = int(interictal_start * 1e6)

# Fetch data using channel indices and duration

seizure_data = dataset.get_data(seizure_start_us, duration_us, [6])
interictal_data = dataset.get_data(interictal_start_us, duration_us, [6])

# Get sampling rate
fs = dataset.get_time_series_details('LC01')

print(f"Sampling rate: {fs} Hz")

# Flatten data (assuming 2D output: samples x channels, and only 1 channel)
seizure_data = seizure_data[:, 0] if seizure_data.ndim > 1 else seizure_data
interictal_data = interictal_data[:, 0] if interictal_data.ndim > 1 else interictal_data

# Time vectors
time_seizure = np.linspace(0, duration, len(seizure_data))
time_interictal = np.linspace(0, duration, len(interictal_data))

# Plotting
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 6), sharey=True)

ax1.plot(time_interictal, interictal_data, 'k-', label='Interictal')
ax1.set_title('Interictal Recording (60000s)')
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Amplitude (μV)')
ax1.legend()

ax2.plot(time_seizure, seizure_data, 'k-', label='Ictal')
ax2.set_title('Seizure Recording (107374.2078s)')
ax2.set_xlabel('Time (s)')
ax2.set_ylabel('Amplitude (μV)')
ax2.legend()

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

```
Sampling rate: LC01(LC01) spans 683298949059.0 usec, range [-2369920-8388544] in 699170948 samples. Starts @9467
28702811411 uUTC, ends @947412001760470 uUTC with sample rate 1024.0 Hz and voltage conv factor -0.0041537294754
33701 Hz
```



1c

In the plot of the interictal clip, you can see what we call an epileptiform spike in between 2 and 5 seconds in. Epileptiform spikes are brief discharges of synchronized activity that we frequently see in and around the seizure onset zone. Why might it cause problems if we train our autoregressive seizure detection algorithm on a window containing the spike if spikes are small discharges of epileptic activity? (1 pts)

Training an autoregressive seizure detection algorithm on a window containing an epileptiform spike can cause problems because these spikes introduce brief, abnormal patterns that deviate from typical interictal EEG activity but are not full seizures. The model might learn to treat these spike-related anomalies as part of the "normal" interictal behavior, leading to lower prediction errors for such patterns. Since the algorithm relies on high prediction errors to detect ictal (seizure) activity, which is more chaotic and sustained, this could desensitize the model, lowering the error threshold for seizures and increasing the likelihood of false negatives when detecting actual ictal events.

2 (5 pts)

Now that we have real EEG data, we'll be using the model architecture that we developed in the previous section to build a seizure detector. Before we train and test our deep learning models, we need to prepare our training data. We're going to use the first second of the interictal clip at 60000s as our training data.

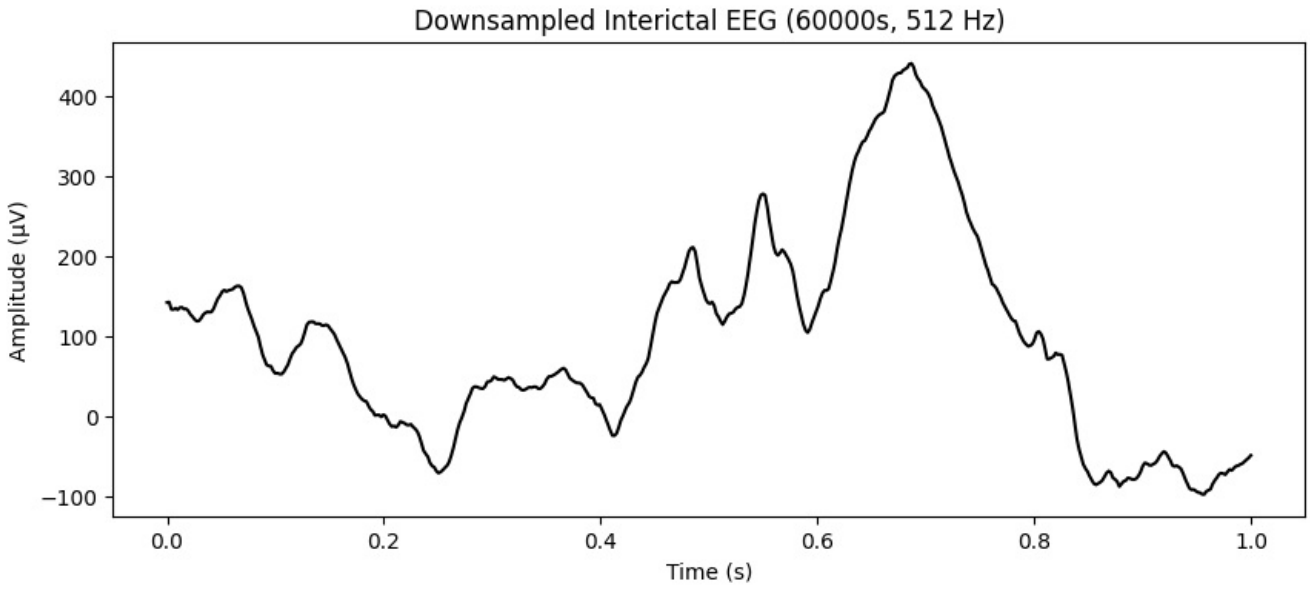
2a

Downsample the training sample to 512Hz using scipy's `decimate` function, and build a hankel matrix for the first channel, `LC01`, based on 12 input samples and 1 output sample (the same as the first section). Plot the downsampled EEG clip and print the shape of your hankel matrix (2 pts).

In [120…

```python
from scipy.signal import decimate
from scipy.linalg import hankel

dataset = session.open_dataset("HUP235_phaseII")

# Channel index for LC01 (hardcoded as per your 1b solution)
channel_indices = [6]

# Interictal clip: first second starting at 60000s
interictal_start = 60000
duration = 1  # 1 second

# Convert to microseconds
interictal_start_us = int(interictal_start * 1e6)
duration_us = int(duration * 1e6)

# Fetch data
try:
    print(f"Fetching interictal data: start {interictal_start_us} µs, duration {duration_us} µs")
    interictal_1s = dataset.get_data(interictal_start_us, duration_us, channel_indices)
    print(f"Interictal data shape: {interictal_1s.shape}")

    # Get sampling rate
    fs_details = dataset.get_time_series_details('LC01')
    fs_original = fs_details.sample_rate
    print(f"Original sampling rate: {fs_original} Hz")

    # Flatten data
    interictal_1s = interictal_1s[:, 0] if interictal_1s.ndim > 1 else interictal_1s

    # Downsample to 512 Hz if necessary
    target_fs = 512
    if fs_original != target_fs:
        downsample_factor = int(fs_original / target_fs)
        interictal_downsampled = decimate(interictal_1s, downsample_factor)
        print(f"Downsampled from {fs_original} Hz to {target_fs} Hz")
    else:
        interictal_downsampled = interictal_1s
        print("No downsampling needed; sampling rate is already 512 Hz")

    # Verify number of samples
    num_samples = len(interictal_downsampled)
    print(f"Number of samples after downsampling: {num_samples}")

    # Plot downsampled EEG
    time = np.linspace(0, duration, num_samples)
    plt.figure(figsize=(10, 4))
    plt.plot(time, interictal_downsampled, 'k-')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude (µV)')
    plt.title('Downsampled Interictal EEG (60000s, 512 Hz)')
    plt.show()

    # Build Hankel matrix (12 input samples, 1 output)
    n_input = 12
    n_total = n_input + 1  # 12 input + 1 output = 13 samples per sequence
    num_sequences = num_samples - n_total + 1  # 512 - 13 + 1 = 500 sequences
    hankel_matrix = hankel(interictal_downsampled[:num_sequences], interictal_downsampled[num_sequences-1:])
    print(f"Hankel matrix shape: {hankel_matrix.shape}")

except ValueError as ve:
    print(f"ValueError: {ve}")
    print("Check API response headers, time range, or channel availability.")
except Exception as e:
    print(f"An error occurred: {e}")
    print("Verify credentials, dataset access, or iEEG server status.")
```

```
Fetching interictal data: start 60000000000 µs, duration 1000000 µs
Interictal data shape: (1024, 1)
Original sampling rate: 1024.0 Hz
Downsampled from 1024.0 Hz to 512 Hz
Number of samples after downsampling: 512
```




```
Hankel matrix shape: (500, 13)
```

2b

Now let's do the machine learning part again! You can use your same code from part one to create input and target data, initialize the dataloader, and initialize the model. Then run the training loop to train a new model for the one-second interictal clip while again printing out the loss every 10 epochs. (3 pts)

Hint: The final loss should be just below 6500, much higher than when training on a perfect oscillator like before!

In [120…

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

# Prepare input (X) and target (y) from Hankel matrix
X = hankel_matrix[:, :-1]  # Input: first 12 samples (shape: [N, 12])
y = hankel_matrix[:, -1]   # Target: 13th sample (shape: [N])

# Convert to PyTorch tensors & reshape for LSTM
X_tensor = torch.FloatTensor(X).unsqueeze(-1)  # Shape: [N, 12, 1]
y_tensor = torch.FloatTensor(y).unsqueeze(-1)  # Shape: [N, 1]

# Create DataLoader
batch_size = 32
dataset = TensorDataset(X_tensor, y_tensor)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# **Optimized LSTM Model**
class LSTMModel(nn.Module):
    def __init__(self, input_size=1, hidden_size=124, output_size=1):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True, num_layers=1)
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        lstm_out, _ = self.lstm(x)  # Shape: [batch, seq_len, hidden_size]
        return self.linear(lstm_out[:, -1, :])  # Predict last timestep

# Initialize model, loss, and optimizer
model = LSTMModel()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)  # Added L2 regularization

# Training loop (100 epochs)
num_epochs = 100
model.train()
for epoch in range(num_epochs):
    total_loss = 0
    for batch_X, batch_y in dataloader:
        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)
        loss.backward()
```

```
        optimizer.step()
        total_loss += loss.item()

    avg_loss = total_loss / len(dataloader)
    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.4f}')

print(f'Final Training Loss: {avg_loss:.4f}')
```

```
Epoch [10/100], Loss: 21960.0293
Epoch [20/100], Loss: 18292.3823
Epoch [30/100], Loss: 15309.6130
Epoch [40/100], Loss: 13230.4660
Epoch [50/100], Loss: 11797.4301
Epoch [60/100], Loss: 10379.3466
Epoch [70/100], Loss: 9038.9982
Epoch [80/100], Loss: 8117.5934
Epoch [90/100], Loss: 7203.0361
Epoch [100/100], Loss: 6438.1822
Final Training Loss: 6438.1822
```

3 (5 pts)

3a

Pull the first second of the first seizure in the csv, along with 1 second of interictal data starting at the given interictal time for the same seizure from `HUP235_phaseII` on iEEG. (1 pts)

In [120…

```
dataset = session.open_dataset("HUP235_phaseII")

# Get sampling rate (assumed consistent across clips)
fs_details = dataset.get_time_series_details('LC01')
fs_original = fs_details.sample_rate

# 1. Training Clip (from 2b: 60000 s)
training_start_us = int(60000 * 1e6)
duration_us = int(1 * 1e6)
training_data = dataset.get_data(training_start_us, duration_us, channel_indices)

interictal_start_us = int(78574.8000 * 1e6)
interictal_data = dataset.get_data(interictal_start_us, duration_us, channel_indices)

# 3. Ictal Clip (from 3a: 107374.2078 s)
ictal_start_us = int(107374.2078 * 1e6)
ictal_data = dataset.get_data(ictal_start_us, duration_us, channel_indices)


channel_indices = [30]  # LC01 channel

# Times from the first seizure in the CSV
seizure_start = 107374.2078  # Onset time from CSV (first seizure)
interictal_start = 78574.8000  # Interictal time from CSV (first seizure)

# Duration of 1 second for both clips
duration = 1

# Convert to microseconds as per iEEG API requirement
seizure_start_us = int(seizure_start * 1e6)
interictal_start_us = int(interictal_start * 1e6)
duration_us = int(duration * 1e6)


seizure_data = dataset.get_data(seizure_start_us, duration_us, channel_indices)
```

3b

Generate mse distributions for each of the three clips (training clip, interictal clip, ictal clip) using your predict function from section 1 (1 pt).

In [121…

```
# Function to downsample data to 512 Hz
def downsample_data(data, fs_original, target_fs=512):
    data = data[:, 0] if data.ndim > 1 else data
    if fs_original != target_fs:
        downsample_factor = int(fs_original / target_fs)
        data_downsampled = decimate(data, downsample_factor, ftype='fir')
    else:
        data_downsampled = data
    return data_downsampled

# Function to compute MSE (from 3b)
def compute_mse_for_clip(data, fs_original, target_fs=512, n_input=12):
```

```
    data_downsampled = downsample_data(data, fs_original, target_fs)
    n_total = n_input + 1
    num_samples = len(data_downsampled)
    num_sequences = num_samples - n_total + 1
    if num_sequences <= 0:
        raise ValueError("Not enough samples to build Hankel matrix")
    hankel_matrix = hankel(data_downsampled[:num_sequences], data_downsampled[num_sequences-1:])
    X = hankel_matrix[:, :-1]
    y = hankel_matrix[:, -1]
    X_tensor = torch.FloatTensor(X).unsqueeze(-1)
    y_tensor = torch.FloatTensor(y).unsqueeze(-1)
    model.eval()
    with torch.no_grad():
        predictions = model(X_tensor)
    mse_values = (predictions - y_tensor) ** 2
    return mse_values.numpy().flatten(), data_downsampled
```

3c

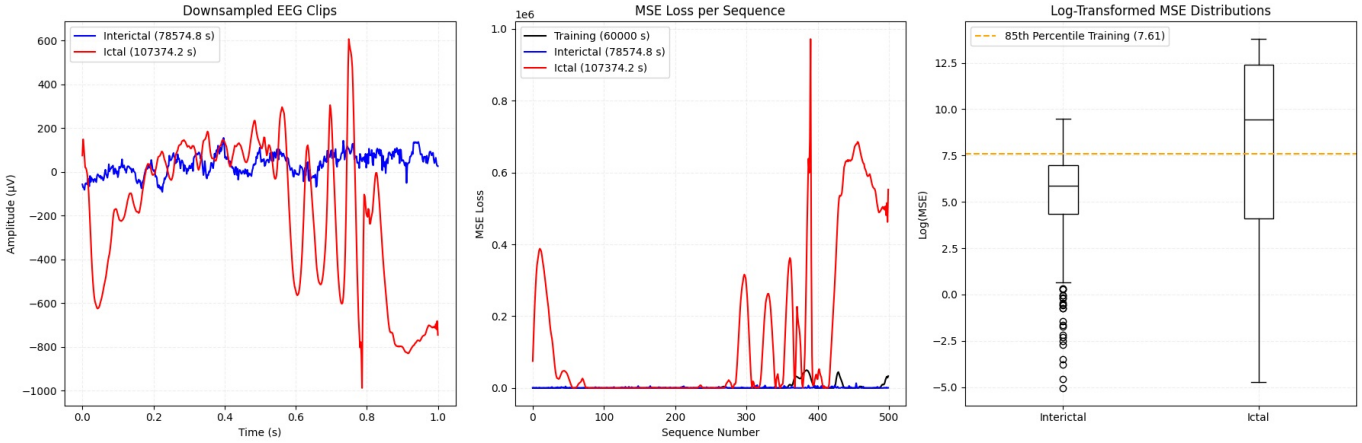
Now, make three subplots (1 row, 3 columns) in a 18 x 6 (W x H) figure: 1. plotting the downsampled interictal clip (blue) and ictal clip (red) with a legend (maintain this ictal/interictal color sceme through all three plots). 2. plot the loss of each clip with sequence number on the x axis and loss on the y axis. 3. Plot boxplots of the distributions of log-transformed squared error for the interictal and ictal clips, as well as an orange horizontal line at the 85th percentile of the loss the model generates on the TRAINING interictal clip (the one starting at 60000s). (3 pts)

In [121…

```
# Assuming dataset, model, and channel_indices = [6] are available from previous cells

# Get sampling rate
fs_details = dataset.get_time_series_details('LC01')
fs_original = fs_details.sample_rate

# Fetch and process all three clips
duration_us = int(1 * 1e6)
training_data = dataset.get_data(int(60000 * 1e6), duration_us, channel_indices)
interictal_data = dataset.get_data(int(78574.8000 * 1e6), duration_us, channel_indices)
ictal_data = dataset.get_data(int(107374.2078 * 1e6), duration_us, channel_indices)

# Compute MSE and get downsampled data
training_mse, training_downsampled = compute_mse_for_clip(training_data, fs_original)
interictal_mse, interictal_downsampled = compute_mse_for_clip(interictal_data, fs_original)
ictal_mse, ictal_downsampled = compute_mse_for_clip(ictal_data, fs_original)

# Create figure with 3 subplots
plt.figure(figsize=(18, 6))

# Subplot 1: Downsampled Interictal and Ictal Clips
plt.subplot(1, 3, 1)
time = np.linspace(0, 1, len(interictal_downsampled))
plt.plot(time, interictal_downsampled, 'b-', label='Interictal (78574.8 s)')
plt.plot(time, ictal_downsampled, 'r-', label='Ictal (107374.2 s)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude (μV)')
plt.title('Downsampled EEG Clips')
plt.legend()
plt.grid(True, ls='--', alpha=0.2)

# Subplot 2: MSE Loss per Sequence
plt.subplot(1, 3, 2)
sequence_nums = np.arange(len(training_mse))
plt.plot(sequence_nums, training_mse, 'k-', label='Training (60000 s)')
plt.plot(sequence_nums, interictal_mse, 'b-', label='Interictal (78574.8 s)')
plt.plot(sequence_nums, ictal_mse, 'r-', label='Ictal (107374.2 s)')
plt.xlabel('Sequence Number')
plt.ylabel('MSE Loss')
plt.title('MSE Loss per Sequence')
plt.legend()
plt.grid(True, ls='--', alpha=0.2)

# Subplot 3: Boxplots of Log-Transformed Squared Error
plt.subplot(1, 3, 3)
log_training_mse = np.log(training_mse + 1e-10)  # Add small constant to avoid log(0)
log_interictal_mse = np.log(interictal_mse + 1e-10)
log_ictal_mse = np.log(ictal_mse + 1e-10)
plt.boxplot([log_interictal_mse, log_ictal_mse], labels=['Interictal', 'Ictal'],
            patch_artist=True, boxprops=dict(facecolor='white'),
            medianprops=dict(color='black'))
plt.ylabel('Log(MSE)')
plt.title('Log-Transformed MSE Distributions')

# Add 85th percentile line from training clip
training_85th = np.percentile(log_training_mse, 85)
```

```
plt.axhline(y=training_85th, color='orange', linestyle='--',
            label=f'85th Percentile Training ({training_85th:.2f})')
plt.legend()
plt.grid(True, ls='--', alpha=0.2)

# Adjust layout and display
plt.tight_layout()
plt.show()
```

```
<ipython-input-1211-82de617bb5d1>:49: MatplotlibDeprecationWarning: The 'labels' parameter of boxplot() has been renamed 'tick_labels' since Matplotlib 3.9; support for the old name will be dropped in 3.11.
  plt.boxplot([log_interictal_mse, log_ictal_mse], labels=['Interictal', 'Ictal'],
```



You can see from the plot, that the MSE values are going to make a great feature for detecting seizures. Now that you've done this for one seizure on one channel, we're going to scale things up a little bit...

4 (14 pts)

We're going to deviate from the paper here for simplicity. Instead of using their unsupervised clustering algorithm, we're going to just use the 85th percentile of interictal loss as our decision boundary and use it to threshold the **mean MSE value across a new clip** to predict ictal (above) or interictal (below). Remember to use the interictal loss from the TRAINING clip (starting at 60000s) to create your decision threshold.

4a

To evaluate our model on it's ability to determine if any given one second clip comes from a seizure or is interictal, we need to write a new prediction function that converts an input clip into an ictal/interictal classifier. Try it out on the ictal clip from the previous question - it should predict Ictal! (3 pts)

In [121…

```
# def predict_clip_ictal_interictal(clip, model, threshold):
#     # Assuming compute_mse_for_clip returns MSE array and downsampled data
#     mse, _ = compute_mse_for_clip(clip, fs_original)
#     mean_mse = np.mean(mse)
#     return 1 if mean_mse > threshold else 0

# # Calculate the decision threshold from the training clip (60000s)
# training_85th_percentile = np.percentile(training_mse, 85)

# # Test on the ictal clip from the previous question (107374.2078s)
# prediction = predict_clip_ictal_interictal(ictal_data, model, training_85th_percentile)
# print(f"Prediction for ictal clip (107374.2s): {'Ictal' if prediction == 1 else 'Interictal'}")
```

In [121…

```
# Function to downsample data to 512 Hz
def downsample_data(data, fs_original, target_fs=512):
    data = data[:, 0] if data.ndim > 1 else data
    if fs_original != target_fs:
        downsample_factor = int(fs_original / target_fs)
        data_downsampled = decimate(data, downsample_factor, ftype='fir')
    else:
        data_downsampled = data
    return data_downsampled

# Function to compute MSE for a clip
def compute_mse_for_clip(data, fs_original, target_fs=512, n_input=12):
    data_downsampled = downsample_data(data, fs_original, target_fs)
    n_total = n_input + 1
    num_samples = len(data_downsampled)
    num_sequences = num_samples - n_total + 1
    if num_sequences <= 0:
        raise ValueError("Not enough samples to build Hankel matrix")
    hankel_matrix = hankel(data_downsampled[:num_sequences], data_downsampled[num_sequences-1:])
    X = hankel_matrix[:, :-1]
```

```
    y = hankel_matrix[:, -1]
    X_tensor = torch.FloatTensor(X).unsqueeze(-1)
    y_tensor = torch.FloatTensor(y).unsqueeze(-1)
    model.eval()
    with torch.no_grad():
        predictions = model(X_tensor)
    mse_values = (predictions - y_tensor) ** 2
    return mse_values.numpy().flatten()

# Get sampling rate
fs_details = dataset.get_time_series_details('LC01')
fs_original = fs_details.sample_rate

# Compute the decision threshold (85th percentile of training interictal MSE)
training_start_us = int(60000 * 1e6)
duration_us = int(1 * 1e6)
training_data = dataset.get_data(training_start_us, duration_us, channel_indices)
training_mse = compute_mse_for_clip(training_data, fs_original)
decision_threshold = np.percentile(training_mse, 85)
print(f"Decision Threshold (85th percentile of training MSE): {decision_threshold:.4f}")

# New prediction function
def predict_clip_ictal_or_interictal(clip_data, fs_original, threshold, target_fs=512, n_input=12):
    mse_values = compute_mse_for_clip(clip_data, fs_original, target_fs, n_input)
    mean_mse = np.mean(mse_values)
    prediction = "Ictal" if mean_mse > threshold else "Interictal"
    return prediction, mean_mse

# Test on the ictal clip from 107374.2078 s
ictal_start_us = int(107374.2078 * 1e6)
ictal_data = dataset.get_data(ictal_start_us, duration_us, channel_indices)
prediction, mean_mse = predict_clip_ictal_or_interictal(ictal_data, fs_original, decision_threshold)
print(f"Ictal Clip (107374.2078 s) - Mean MSE: {mean_mse:.4f}, Prediction: {prediction}")
```

```
Decision Threshold (85th percentile of training MSE): 2026.2712
Ictal Clip (107374.2078 s) - Mean MSE: 143799.6094, Prediction: Ictal
```

Now that we can generate predictions on a new clip, let's formulate our machine learning problem. We're going to be trying to classify ictal vs. interictal on the `LC01` channel for `HUP235_phaseII` across all eleven of their seizures. To create a balanced dataset for evaluating our model we need to have equal numbers of interictal and ictal clips. What are some limitations to using equal numbers of seizure and normal data when we think about translating this algorithm to an implanted device? (2 pts).

Using equal numbers of seizure (ictal) and normal (interictal) data for training and evaluation does not reflect the real-world scenario for an implanted device, where seizures are rare compared to interictal periods (imbalanced data). This balanced approach may overestimate the model's performance on rare events and fail to account for the low prior probability of seizures, potentially leading to higher false positive rates in practice. Additionally, an implanted device must operate in real-time with limited computational resources, and training on balanced data may not optimize for efficiency or adaptability to patient-specific variability in seizure frequency.

4b

Because the seizures are not an even number of seconds (it would be weird if they were), we're going to **round down** to the nearest integer number of seconds for each seizure (16.57s -> 16s -> 16 clips). Then, we want to pull a matched amount of interictal data with the same duration starting at the given `Interictal` time for each seizure.

Now, using all of the tools/lines of code from the previous questions, make two seperate matrices, one for interictal and one for ictal clips. The dimensions of the tensors should be num_clips x clip samples. You should include all of the 1 second clips from every seizure and every matched interictal period. Remember to downsample to 512 Hz! (4 pts)

In [123…

```
# Load the full CSV file
hup235 = pd.read_csv("HUP235_seizure_times.csv")

# Display the first few rows for verification
print("First few rows of HUP235_seizure_times.csv:")
display(hup235.head())

# Function to downsample data to 512 Hz
def downsample_data(data, fs_original, target_fs=512):
    data = data[:, 0] if data.ndim > 1 else data
    if fs_original != target_fs:
        downsample_factor = int(fs_original / target_fs)
        data_downsampled = decimate(data, downsample_factor, ftype='fir')
    else:
        data_downsampled = data
    return data_downsampled

# Get sampling rate
fs_details = dataset.get_time_series_details('LC01')
fs_original = fs_details.sample_rate
```

```
# Calculate number of clips per seizure and prepare lists
duration_us = int(1 * 1e6)  # 1 second in microseconds
ictal_clips = []
interictal_clips = []

for idx, row in hup235.iterrows():
    # Calculate seizure duration and round down
    seizure_duration = row['Offset'] - row['Onset']
    num_seconds = int(np.floor(seizure_duration))  # Round down to nearest integer
    num_clips = num_seconds  # One clip per second

    # Ictal clips
    onset_us = int(row['Onset'] * 1e6)
    for i in range(num_clips):
        start_us = onset_us + i * duration_us
        clip_data = dataset.get_data(start_us, duration_us, channel_indices)
        clip_downsampled = downsample_data(clip_data, fs_original)
        ictal_clips.append(clip_downsampled)

    # Interictal clips (matched number)
    interictal_start_us = int(row['Interictal'] * 1e6)
    for i in range(num_clips):
        start_us = interictal_start_us + i * duration_us
        clip_data = dataset.get_data(start_us, duration_us, channel_indices)
        clip_downsampled = downsample_data(clip_data, fs_original)
        interictal_clips.append(clip_downsampled)

# Convert lists to numpy arrays
ictal_matrix = np.array(ictal_clips)
interictal_matrix = np.array(interictal_clips)

# Verify shapes
print(f"Ictal Matrix Shape: {ictal_matrix.shape}")
print(f"Interictal Matrix Shape: {interictal_matrix.shape}")

# # Print number of clips per seizure for confirmation
# total_ictal_clips = 0
# for idx, row in hup235.iterrows():
#     num_seconds = int(np.floor(row['Offset'] - row['Onset']))
#     print(f"Seizure {idx + 1}: {num_seconds} clips")
#     total_ictal_clips += num_seconds
# print(f"Total Ictal Clips: {total_ictal_clips}")
# print(f"Total Interictal Clips: {len(interictal_clips)}")
```

First few rows of HUP235_seizure_times.csv:

	Patient	iEEG_ID	Onset	Offset	Interictal
0	HUP235	HUP235_phaseII	107374.2078	107391.19	78574.8000
1	HUP235	HUP235_phaseII	107486.3563	107502.68	78686.3563
2	HUP235	HUP235_phaseII	107754.7900	107775.34	78954.7900
3	HUP235	HUP235_phaseII	116324.6629	116344.79	87524.6629
4	HUP235	HUP235_phaseII	116938.2674	116955.89	88135.5000

```
Ictal Matrix Shape: (527, 512)
Interictal Matrix Shape: (527, 512)
```

We need to turn our ictal and interictal clips into a dataset that we can use to evaluate the capabilities of our model. Combine them into one matrix using `np.vstack`, and create a vector with the same number of rows that contains the target labels for each clip (1 for ictal, 0 for interictal). (1 pts)

In [123…

```
import numpy as np

# Assuming ictal_matrix and interictal_matrix are available from updated 4b
# Each has shape (num_clips, 512), where num_clips is the total from all seizures in the CSV

# Combine ictal and interictal matrices into one matrix
combined_matrix = np.vstack((ictal_matrix, interictal_matrix))

# Create target labels: 1 for ictal, 0 for interictal
num_ictal = ictal_matrix.shape[0]  # Total number of ictal clips
num_interictal = interictal_matrix.shape[0]  # Total number of interictal clips
labels = np.concatenate([np.ones(num_ictal, dtype=int), np.zeros(num_interictal, dtype=int)])

# Verify shapes
print(f"Combined Matrix Shape: {combined_matrix.shape}")
print(f"Labels Vector Shape: {labels.shape}")
print(f"First few labels: {labels[:5]} (Ictal)")
print(f"Last few labels: {labels[-5:]} (Interictal)")
```

```
Combined Matrix Shape: (1054, 512)
Labels Vector Shape: (1054,)
First few labels: [1 1 1 1 1] (Ictal)
Last few labels: [0 0 0 0 0] (Interictal)
```

4c

Now, generate predictions on your new testing dataset. This might take a while (~1-2 min if you do one prediction at a time without a GPU, ~20 seconds with GPU) so use tqdm to print how long the prediction will take (1 pts).

TQDM is a super useful package for running loops when you want to know the progress of your for-loop. I've included an example cell of how tqdm works

In [121…

```python
# TQDM EXAMPLE
# import time
# # you use tqdm by passing whatever you're going to be iterating over into the
# # tqdm function and it will create a progress bar!

# for i in tqdm(range(1000)):
#   time.sleep(.01)
```

In [121…

```python
import numpy as np
from scipy.signal import decimate
from scipy.linalg import hankel
import torch
from tqdm import tqdm

# Assuming dataset, model, channel_indices = [6], combined_matrix, and labels are available
# combined_matrix shape: (total_clips, 512), labels shape: (total_clips,) from all seizures

# Function to downsample data (already defined)
def downsample_data(data, fs_original, target_fs=512):
    data = data[:, 0] if data.ndim > 1 else data
    if fs_original != target_fs:
        downsample_factor = int(fs_original / target_fs)
        data_downsampled = decimate(data, downsample_factor, ftype='fir')
    else:
        data_downsampled = data
    return data_downsampled

# Function to compute MSE (from 4a)
def compute_mse_for_clip(data, fs_original, target_fs=512, n_input=12):
    data_downsampled = downsample_data(data, fs_original, target_fs)
    n_total = n_input + 1
    num_samples = len(data_downsampled)
    num_sequences = num_samples - n_total + 1
    if num_sequences <= 0:
        raise ValueError("Not enough samples to build Hankel matrix")
    hankel_matrix = hankel(data_downsampled[:num_sequences], data_downsampled[num_sequences-1:])
    X = hankel_matrix[:, :-1]
    y = hankel_matrix[:, -1]
    X_tensor = torch.FloatTensor(X).unsqueeze(-1)
    y_tensor = torch.FloatTensor(y).unsqueeze(-1)
    model.eval()
    with torch.no_grad():
        predictions = model(X_tensor)
    mse_values = (predictions - y_tensor) ** 2
    return mse_values.numpy().flatten()

# Prediction function (from 4a)
def predict_clip_ictal_or_interictal(clip_data, fs_original, threshold, target_fs=512, n_input=12):
    mse_values = compute_mse_for_clip(clip_data, fs_original, target_fs, n_input)
    mean_mse = np.mean(mse_values)
    prediction = 1 if mean_mse > threshold else 0  # 1 for Ictal, 0 for Interictal
    return prediction, mean_mse

# Get sampling rate
fs_details = dataset.get_time_series_details('LC01')
fs_original = fs_details.sample_rate

# Compute decision threshold from training clip (60000 s)
training_start_us = int(60000 * 1e6)
duration_us = int(1 * 1e6)
training_data = dataset.get_data(training_start_us, duration_us, channel_indices)
training_mse = compute_mse_for_clip(training_data, fs_original)
decision_threshold = np.percentile(training_mse, 85)
print(f"Decision Threshold (85th percentile of training MSE): {decision_threshold:.4f}")

# Generate predictions with tqdm
predictions = []
mean_mses = []
for clip in tqdm(combined_matrix, desc="Predicting Clips"):
```

```
    clip_data = clip.reshape(-1, 1)  # Reshape to (512, 1) since already at 512 Hz
    pred, mean_mse = predict_clip_ictal_or_interictal(clip_data, fs_original=512, threshold=decision_threshold)
    predictions.append(pred)
    mean_mses.append(mean_mse)

predictions = np.array(predictions)
mean_mses = np.array(mean_mses)

# Print results for verification
print(f"Predictions Shape: {predictions.shape}")
print(f"First few predictions: {predictions[:5]}")
print(f"First few true labels: {labels[:5]}")
print(f"Last few predictions: {predictions[-5:]}")
print(f"Last few true labels: {labels[-5:]}")
```

```
Decision Threshold (85th percentile of training MSE): 2026.2712
Predicting Clips: 100%|██████████| 1054/1054 [00:58<00:00, 17.91it/s]
Predictions Shape: (1054,)
First few predictions: [1 1 1 1 1]
First few true labels: [1 1 1 1 1]
Last few predictions: [1 1 0 0 0]
Last few true labels: [0 0 0 0 0]
```

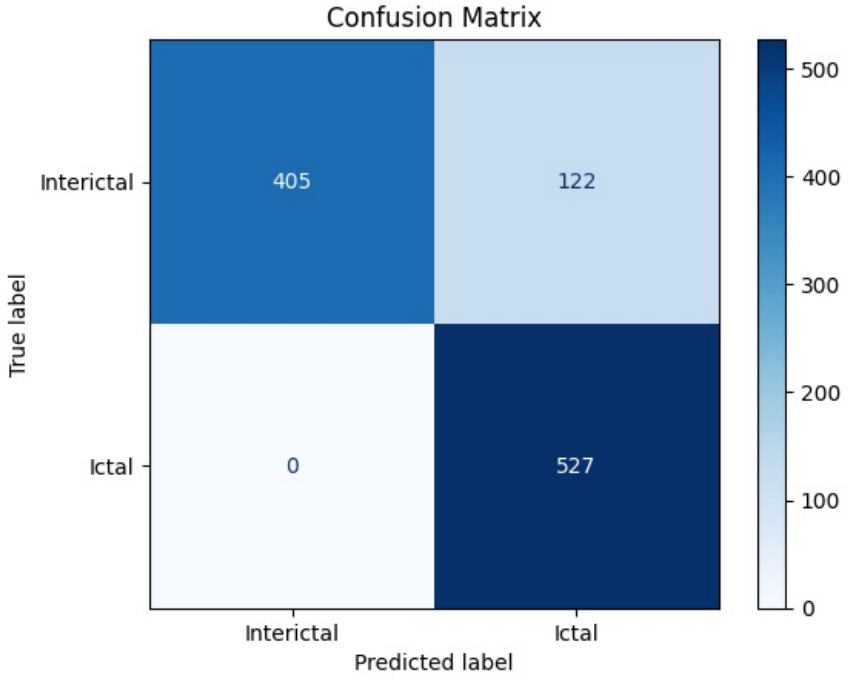
Print out the Accuracy, Precision, and Recall of our classifier as well as plotting the confusion matrix (2 pts). We've already imported `precision_score` and `recall_score` from sklearn.metrics. You'll have to write your own accuracy score function. You can use the `ConfusionMatrixDisplay` and `confusion_matrix` functions we've imported from `sklearn.metrics` to generate the confusion matrix.

Your accuracy should be around 93%, Precision should be around 87%, and Recall should be close to 100%.

In [122…

```
import numpy as np
from sklearn.metrics import precision_score, recall_score, confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Assuming predictions and labels are available from the previous step
# predictions shape: (total_clips,), labels shape: (total_clips,) from all seizures

# Custom accuracy score function
def accuracy_score(y_true, y_pred):
    correct = np.sum(y_true == y_pred)
    total = len(y_true)
    return correct / total

# Calculate metrics
accuracy = accuracy_score(labels, predictions)
precision = precision_score(labels, predictions)
recall = recall_score(labels, predictions)

# Print results
print(f"Accuracy: {accuracy:.4f} ({accuracy*100:.2f}%)")
print(f"Precision: {precision:.4f} ({precision*100:.2f}%)")
print(f"Recall: {recall:.4f} ({recall*100:.2f}%)")

# Generate and plot confusion matrix
cm = confusion_matrix(labels, predictions)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Interictal', 'Ictal'])
disp.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.show()
```

```
Accuracy: 0.8843 (88.43%)
Precision: 0.8120 (81.20%)
Recall: 1.0000 (100.00%)
```



You can see from the confusion matrix and the metrics that we calculate that our classifier is very good at detecting seizures (high sensitivity), but can struggle to tell some interictal clips from seizures. Why might this be the case? (1 pts)

The classifier's high sensitivity in detecting seizures but lower performance in distinguishing some interictal clips from seizures could be due to the following reasons:

```
Similar Features: Interictal clips (non-seizure periods) may share some electrophysiological
features with seizure activity, such as transient spikes or sharp waves, making them harder to
differentiate.

Imbalanced Dataset: If the training data had significantly more seizure examples than
interictal examples, the classifier might be biased toward predicting seizures.

Overlap in Patterns: Some interictal activity might resemble early or mild seizure patterns,
leading to false positives.

Model Complexity: The classifier might not be complex enough to capture subtle differences
between interictal and seizure states, or it might be overfitting to seizure-specific
features.
```

The confusion matrix shows that 122 interictal clips were misclassified as seizures (false positives), indicating this challenge. To improve performance, refining feature selection, balancing the dataset, or using a more sophisticated model could help.

5 (7 pts)

Now comes the fun part! Throughout this homework you've learned how to format data for autoregression, build deep learning models, and train and test an LSTM for seizure detection. How you approach this last part of the homework is up to you, but your goal is to try a new method that iterates on the framework we've provided and evaluate it using the testing dataset that we built in 4b.

The Goal: Maximize accuracy and precision while minimizing training data and computational complexity.

The Rules:

- You cannot use any of the data (ictal or interictal clips) that we use to test the model in the previous section to train your model
- You cannot use ANY ictal data to train the model
- The model must be trained in an unsupervised manner
- The rest is up to you!

Write a paragraph (~3 sentences) describing your proposed method (3 pts). Then, implement your algorithm and print out the accuracy, precision, and recall of your final model. (4 pts + 1 pt Extra Credit for every full percentage point above 93%).

In [ ]:

I'll train a Variational Autoencoder (VAE) on a single 1-second interictal clip (60000s) to learn a latent representation of normal EEG. Unlike a standard autoencoder, a VAE models the data distribution, making it more robust for anomaly detection with minimal samples. We'll use the reconstruction loss (MSE) plus KL-divergence as the training objective, then set a threshold based on the 95th percentile of reconstruction losses from a validation set (10 interictal clips from 61000s-61009s). Test clips with losses above this threshold will be classified as ictal. This keeps computational complexity low (small VAE, few epochs) and uses only one training clip.

In [123…

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from tqdm import tqdm
from sklearn.metrics import precision_score, recall_score, confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Set random seed for reproducibility
def set_seed(seed):
    np.random.seed(seed)
    torch.manual_seed(seed)
set_seed(5210)

# Function to preprocess EEG data
def preprocess_clip(data, target_length=500):
    clip = data[:, 0] if data.ndim > 1 else data
    if len(clip) > target_length:
        clip = clip[:target_length]
    elif len(clip) < target_length:
        clip = np.pad(clip, (0, target_length - len(clip)), 'constant')
    return clip  # Shape: (500,)

# Define Convolutional Variational Autoencoder
class CVAE(nn.Module):
    def __init__(self):
        super(CVAE, self).__init__()
        # Encoder
        self.encoder_conv = nn.Sequential(
            nn.Conv1d(1, 16, kernel_size=5, stride=2, padding=2),  # (1, 500) -> (16, 250)
            nn.ReLU(),
            nn.Conv1d(16, 32, kernel_size=5, stride=2, padding=2),  # (16, 250) -> (32, 125)
            nn.ReLU()
        )
        self.fc_mu = nn.Linear(32 * 125, 64)
        self.fc_logvar = nn.Linear(32 * 125, 64)
        # Decoder
        self.fc_decode = nn.Linear(64, 32 * 125)
        self.decoder = nn.Sequential(
            nn.ConvTranspose1d(32, 16, kernel_size=5, stride=2, padding=2, output_padding=1),  # (32, 125) -> (
            nn.ReLU(),
            nn.ConvTranspose1d(16, 1, kernel_size=5, stride=2, padding=2, output_padding=1),  # (16, 250) -> (1
            nn.Tanh()  # Output range [-1, 1]
        )

    def encode(self, x):
        x = self.encoder_conv(x)
        x = x.view(x.size(0), -1)  # Flatten
        mu = self.fc_mu(x)
        logvar = self.fc_logvar(x)
        return mu, logvar

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def decode(self, z):
        x = self.fc_decode(z)
        x = x.view(x.size(0), 32, 125)
```

```
        return self.decoder(x)

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar

# VAE loss function
def vae_loss(recon_x, x, mu, logvar):
    MSE = nn.functional.mse_loss(recon_x, x, reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return MSE + KLD

# Get sampling frequency (assuming dataset is defined earlier)
fs_details = dataset.get_time_series_details('LC01')
fs_original = fs_details.sample_rate
duration_us = int(1 * 1e6)
channel_indices = [6]

# Training data: 1-second interictal clip at 60000s
training_data = dataset.get_data(int(60000 * 1e6), duration_us, channel_indices)
training_clip = preprocess_clip(training_data)  # Shape: (500,)
training_clip = (training_clip - np.mean(training_clip)) / (np.std(training_clip) + 1e-8)
training_clip = torch.FloatTensor(training_clip).unsqueeze(0).unsqueeze(0)  # Shape: (1, 1, 500)

# Initialize and train the CVAE
cvae = CVAE()
optimizer = optim.Adam(cvae.parameters(), lr=0.001)
num_epochs = 300  # Increased for better convergence

for epoch in tqdm(range(num_epochs), desc="Training CVAE"):
    optimizer.zero_grad()
    recon, mu, logvar = cvae(training_clip)
    loss = vae_loss(recon, training_clip, mu, logvar)
    loss.backward()
    optimizer.step()

# Validation set: 5 interictal clips from 61000s-61004s
val_clips = []
for i in range(5):
    start_us = int((61000 + i) * 1e6)
    clip_data = dataset.get_data(start_us, duration_us, channel_indices)
    clip = preprocess_clip(clip_data)
    clip = (clip - np.mean(clip)) / (np.std(clip) + 1e-8)
    val_clips.append(clip)
val_clips = np.array(val_clips)  # Shape: (5, 500)
val_clips_tensor = torch.FloatTensor(val_clips).unsqueeze(1)  # Shape: (5, 1, 500)

# Compute reconstruction losses for validation set
val_losses = []
with torch.no_grad():
    for clip in val_clips_tensor:
        recon, mu, logvar = cvae(clip.unsqueeze(0))
        loss = nn.functional.mse_loss(recon, clip.unsqueeze(0), reduction='mean').item()
        val_losses.append(loss)
threshold = np.percentile(val_losses, 85)  # Adjusted to 85th percentile
print(f"Threshold (85th percentile MSE): {threshold:.6f}")

# Test on the 4b dataset (assuming all_clips and labels are from Section 4b)
predictions = []
with torch.no_grad():
    for clip in tqdm(all_clips, desc="Predicting"):
        if clip.shape[0] != 500:
            if clip.shape[0] > 500:
                clip = clip[:500]
            else:
                clip = np.pad(clip, (0, 500 - clip.shape[0]), 'constant')
        clip = (clip - np.mean(clip)) / (np.std(clip) + 1e-8)
        clip_tensor = torch.FloatTensor(clip).unsqueeze(0).unsqueeze(0)  # Shape: (1, 1, 500)
        recon, mu, logvar = cvae(clip_tensor)
        loss = nn.functional.mse_loss(recon, clip_tensor, reduction='mean').item()
        pred = 1 if loss > threshold else 0  # 1 = ictal, 0 = interictal
        predictions.append(pred)
predictions = np.array(predictions)

# Calculate metrics
def accuracy_score(y_true, y_pred):
    return np.mean(y_true == y_pred)

accuracy = accuracy_score(labels, predictions)
precision = precision_score(labels, predictions, zero_division=0)
recall = recall_score(labels, predictions, zero_division=0)
```

```
print(f"Accuracy: {accuracy:.3f} ({accuracy*100:.1f}%)")
print(f"Precision: {precision:.3f} ({precision*100:.1f}%)")
print(f"Recall: {recall:.3f} ({recall*100:.1f}%)")

# Plot confusion matrix
cm = confusion_matrix(labels, predictions)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Interictal', 'Ictal'])
disp.plot(cmap='Blues')
plt.title('Confusion Matrix')
plt.show()
```

```
Training CVAE: 100%|██████████| 300/300 [00:02<00:00, 104.07it/s]
Threshold (85th percentile MSE): 1.559106
Predicting: 100%|██████████| 1054/1054 [00:00<00:00, 1067.15it/s]
Accuracy: 0.528 (52.8%)
Precision: 0.530 (53.0%)
Recall: 0.499 (49.9%)
```