CS637A Group 7 Project Presentation

LATTE: LSTM Self-Attention based Anomaly Detection

- **Authors:** VIPIN KUMAR KUKKALA, SOORYAA VIGNESH THIRULOGA, and SUDEEP PASRICHA

- **International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2021**

Members:

- Abhinav Kumar Singh (200017)
- Dhruv Khandelwal (200331)
- Harshit Gupta (200429)
- Paramveer Singh Choudhary (200665)
- Ronit Mittal (200820)
- Dusane Atharv Sachin (200356)

Course Instructor:
Dr. Indranil Saha

Introduction

Automotive Complexity
Modern vehicles are increasingly complex due to the push for autonomous driving.
Intense competition among automakers further complicates Electronic Control Units (ECUs).

ADAS Challenges
Advanced Driver Assistance Systems (ADAS) rely on diverse external systems using various protocols.
Growing connectivity exposes vehicles to potential cyber threats.

Safety Risks
Increased software and hardware complexity poses safety risks.
Cyber-attacks can induce anomalies, affecting critical applications like pedestrian detection and airbag deployment.

Attack Scenarios
Various attacks, including network flooding, pose a threat.
Attacks can lead to catastrophic consequences, emphasizing the need for early detection.

Current Security
Traditional security mechanisms prove inadequate against sophisticated attacks.
Continuous system-level monitoring is crucial for detecting and preventing cyber-attacks.

LATTE - Anomaly Detection Framework

LATTE: LSTM (Long Short-Term Memory) Self-Attention Anomaly Detection

Motivation for LATTE:
Rule-based systems fall short of addressing modern cyber-attack complexity.
An advanced anomaly detection system, like LATTE, is essential for safeguarding automotive systems.

Overview of LATTE:

- **Deep Learning Models:**
  LATTE uses a Long-Short Term Memory (LSTM) predictor with a self-attention mechanism.
  This model learns normal automotive behavior during the design phase.

- **Detection Mechanism:**
  The system deploys a One-Class Support Vector Machine (OCSVM) with the LSTM model to detect anomalies in real-time.

- **System Integration:**
  Proposed modifications to existing vehicle communication controllers facilitate LATTE deployment on Electronic Control Units

- **Deviation Measures Analysis:**
  A comprehensive analysis selects deviation measures that quantify deviations from normal system behavior.

System Overview

ECU (Electrical Control Unit) Components:

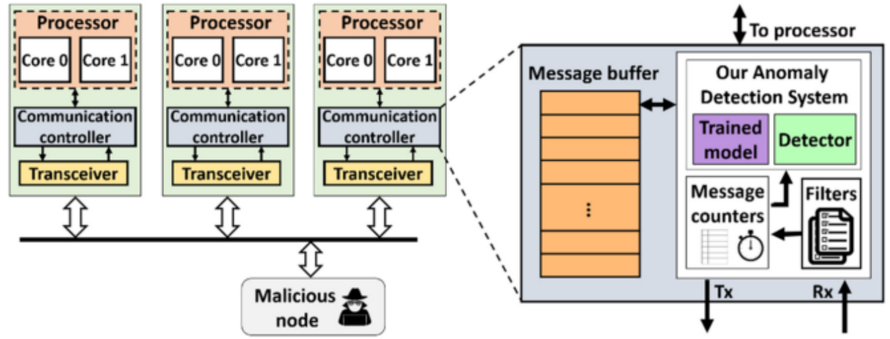
- Each ECU comprises a processor, communication controller, and transceiver.
- Processors execute real-time automotive applications with strict timing constraints.

Communication Controller:

- Facilitates data movement between the processor and the network.
- Key functions include data packing, CAN (Controller Area Network) frame transmission, and message filtering.

Transceiver:

- Serves as an interface between the physical CAN network and the ECU.
- Enables CAN frame transmission and reception.



Anomaly Detection Process:

- Modified CAN communication controllers monitor message frequencies.
- The anomaly detection system uses LSTM-based attention model for prediction.
- Deviation measure from the true message is computed and analyzed using a non-linear classifier.
- Anomalous messages are discarded, preventing execution of attacker messages.

Communication Overview

CAN (Controller Area Network) as In-Vehicle Network Protocol:

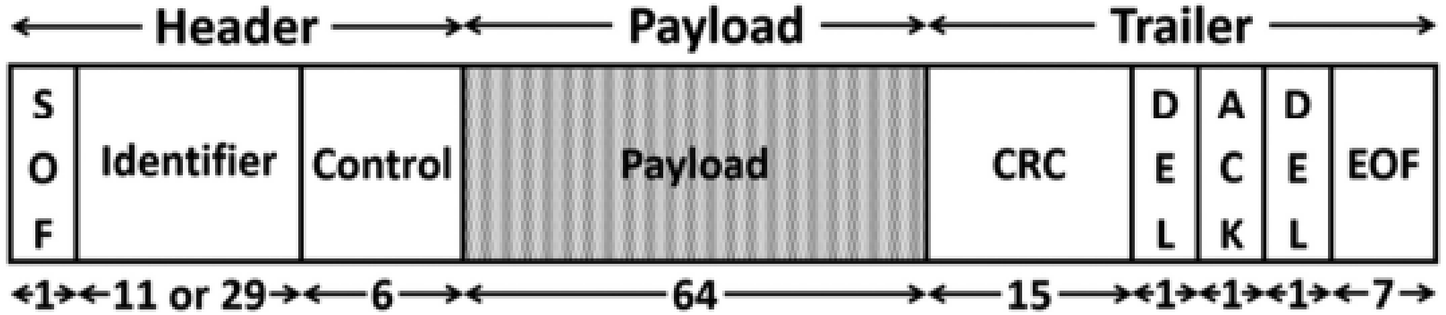
- Consider CAN for exchanging time-critical messages between ECUs.
- Lightweight, low-cost, event-triggered protocol, with CAN 2.0B as the industry standard.

CAN Message Structure:

- Messages consist of signal values grouped in CAN frames.
- CAN frame structure includes header, payload, and trailer segments (figure below).
- Our framework focuses on monitoring the payload field, agnostic to the in-vehicle network protocol.

Framework Operation:

- LATTE framework operates on the payload segment to detect cyber-attacks.
- Learns temporal dependencies between message instances during design time.
- Detects deviations at runtime, facilitating efficient identification of cyber-attacks.



Threat Model

Attacker Access Points:

- Attacker gains access via common threat vectors: OBD-II port, in-vehicle network probing, or connected V2X ADAS systems.
- Assumes access to in-vehicle network parameters for malicious message transmission.

Anomaly Detection Protection:

System safeguards against various cyber-attacks:

- Constant Attack
- Continuous Attack
- Replay Attack
- Dropping Attack
- Distributed Denial of Service (DDoS) Attack

Detection Complexity:

- Complexity of detection varies for each attack type.
- LATTE system extends protection by analyzing abnormal message rates and deviations in the payload field.
- Effective protection against hard-to-detect attacks in the automotive domain.

Proposed LATTE Framework

Data Acquisition

Data Collection from Trusted Vehicle:

- Ensure in-vehicle network and ECUs are free from attackers.
- Cover a diverse range of normal operating conditions.
- Access points: OBD-II port or CAN network probing.

Data Preprocessing:

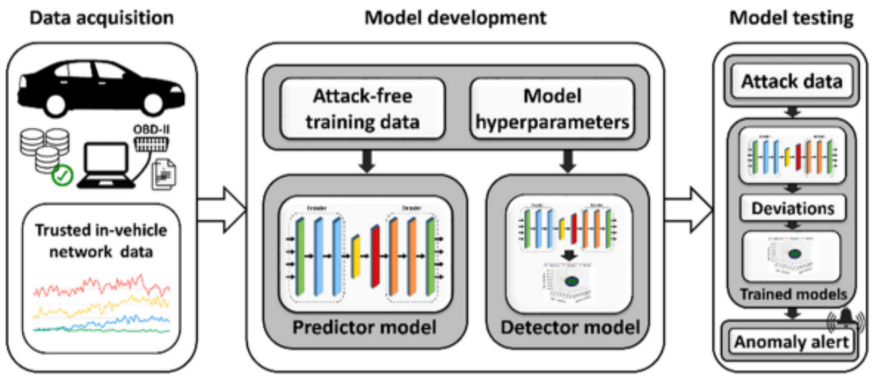
- The dataset grouped by unique CAN message identifiers and each group is processed independently
- Features include timestamp, message identifier, signal values, and label (0 for non-anomalous, 1 for anomalous).
- Signal values scaled between 0 to 1 to address variance.

Labeling for Anomaly Detection Models:

- Label column values set to 0 for training and validation datasets (trusted environment).
- During testing, the label column is set to 1 for samples in the attack interval.
- Not used during training predictor and detector models.

Relative Time Concept:

- Time is considered in a relative manner during training and deployment.
- No dependency on absolute time.



Predictor Model

Predictor and Detector Models:

- Collaborative approach to detect cyber-attacks in the in-vehicle network.
- Predictor model learns normal system behavior through unsupervised learning.
- Detector model identifies anomalies by recognizing deviations from learned patterns.

Predictor Model Operation:

- Learns normal behavior, predicting next message instances accurately during design.
- Utilizes unsupervised learning on non-anomalous data.
- Employs stacked LSTM-based encoder-decoder with a self-attention mechanism.
- Employs a rolling window approach for training, capturing temporal dependencies.

Model Architecture:

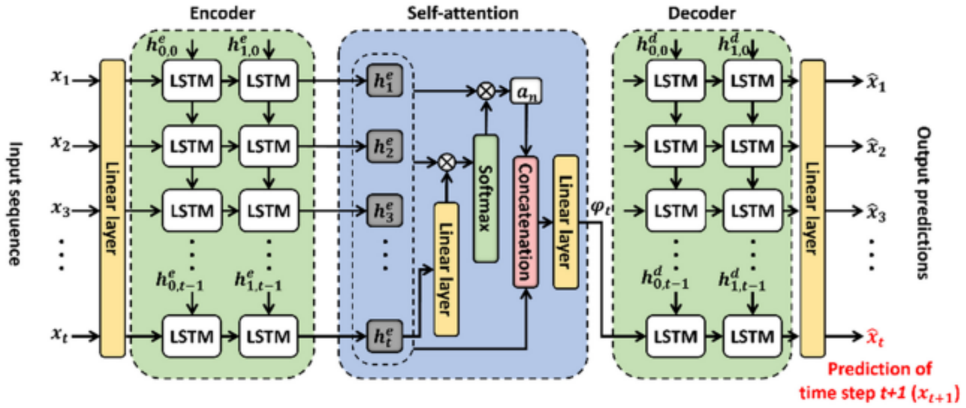
- **Input Layer:**
  Time series CAN message data $X = \{x_1, x_2, \ldots, x_t\}$
  Generates a 128-dimensional embedding for each input.
- **Stacked LSTM Encoder:`**
  Produces a 64-dimension encoder output h_{et} representing temporal relationships.
- **Self-Attention Mechanism:**
  Generates a context vector ϕ_t by considering the importance of hidden states from earlier time steps.

Predictor Model (cont.)

- **Stacked LSTM Decoder:**
  Takes the context vector ϕ_t and the previous decoder's hidden state hd_{t-1} as input.
  Produces a 64-dimension output hd_t that is passed to the last linear layer to obtain a k-dimensional output.
- **Output Layer:**
  Last linear layer obtains a k-dimensional output x'_t, representing signal values of the next message instance

Training Process:

- Unsupervised learning using non-anomalous data.
- Rolling window approach with a fixed-size subsequence.
- Minimizes prediction error using mean square error (MSE) loss function.
- Hyperparameter: Subsequence length.



Training and Validation:

- Dataset split into training (80%) and validation (20%) data without shuffling.
- Continuous adjustment of weights through back-propagation.
- Early stopping mechanism to prevent overfitting.
- Mini-batches used for efficiency.

Detector Model

Separate Classifier (Detector Model):

- Objective: Distinguishing between normal and anomalous messages.
- Challenges:
  - Rapidly growing in-vehicle network data.
  - Imbalanced dataset (few attack samples compared to normal samples).
- Solution: One-Class Support Vector Machine (OCSVM).

OCSVM for Unbalanced Datasets:

- Adaptable for highly imbalanced datasets.
- Constructs the smallest hypersphere containing training data at design time.
- Identifies samples outside the hypersphere as anomalies at runtime.
- Training Process:
  - Input: Normal training dataset and predictor model's deviations.
  - Learns from deviations using absolute values.

Deviation Representation:

- For a message m with k_m signals, the deviation vector $\Delta_{m,t}$ at time step t is given by:

$$\Delta_{m,t} = \left(S'_{i,t} - S_{i,t+1}\right) \in R^2, \forall i \in [1, k_m]$$

- $S'_{i,t}$: Prediction of the next true i^{th} signal value $\left(S_{i,t+1}\right)$ made at time step t .

Detector Model (cont.)

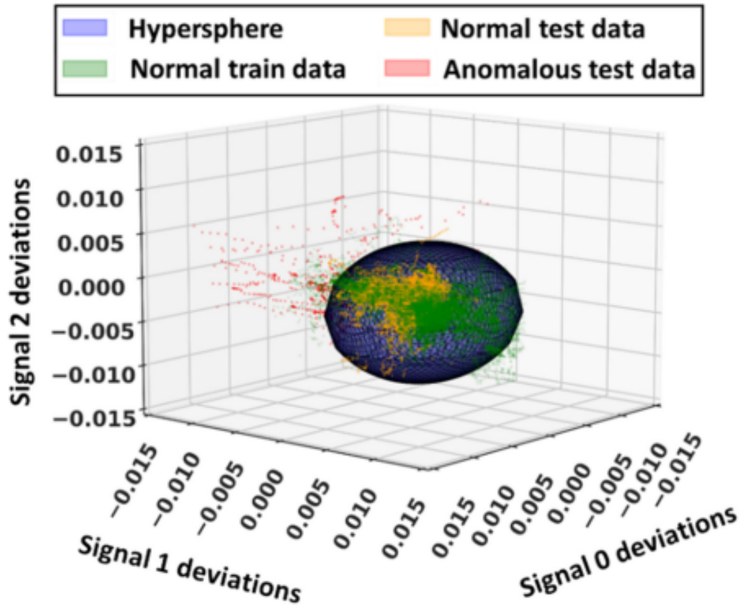
Deviation Variants:

Other deviation measures explored:

- $\Delta_{sum,m,t}$: Sum of absolute signal deviations.
- $\Delta_{avd,m,t}$: Average of absolute signal deviations.
- $\Delta_{max,m,t}$: Maximum absolute signal deviation.

Testing Process:

- OCSVM constructs hypersphere based on predictor's small deviations for normal samples.
- Yellow dots (normal samples) classified within hypersphere; red dots (anomalous samples) outside.
- Predictor and detector models collaborate for effective attack detection.



Model Testing

Test Dataset Composition:

- Consists of:
  - Normal samples (label 0).
  - Anomalous samples representing diverse attacks (label 1).
- Labels guide anomaly classification during testing.

Predictor Model Evaluation:

- Each sample sent to predictor model for next instance prediction.
- Deviation computed based on true message data.

OCSVM Detector Model:

- Receives deviation vectors from predictor model.
- Computes position in k-dimensional space (k = number of signals).
- Classification:
  - Non-anomalous if inside the learned hypersphere.
  - Anomalous if outside hypersphere.
- Anomaly alerts raised for further action.

Anomaly Detection System Deployment

Centralized Approach:

- A single powerful ECU monitors the entire CAN bus.
- Vulnerable to single-point failures.
- Prone to communication issues during extreme scenarios like DDoS attacks.
- Susceptible to undetected attacks if the centralized ECU is compromised.

Distributed Approach:

- Anomaly detection is distributed across multiple ECUs.
- Advantages:
  - Resilient to single-point failures.
  - Improved communication during network congestion.
  - Harder for attackers to compromise the entire system.
  - Faster response time in detecting anomalies.
  - Lower computation load with split detection tasks.
  - Monitoring is limited to relevant messages, reducing overhead.

Memory Optimization:

- Use of a circular buffer to store previous normal samples and predictions.
- Circular buffer size configured at design time (subsequence length).
- Minimizes storage overhead while preserving data dependencies.

Experimental Analysis

LATTE Variants:

- Explored five variants: LATTE-ST, LATTE-Diff, LATTE-Sum, LATTE-Avg, LATTE-Max.
- Different deviation criteria for OCSVM detection.
- LATTE-Diff selected as the candidate model for subsequent experiments.

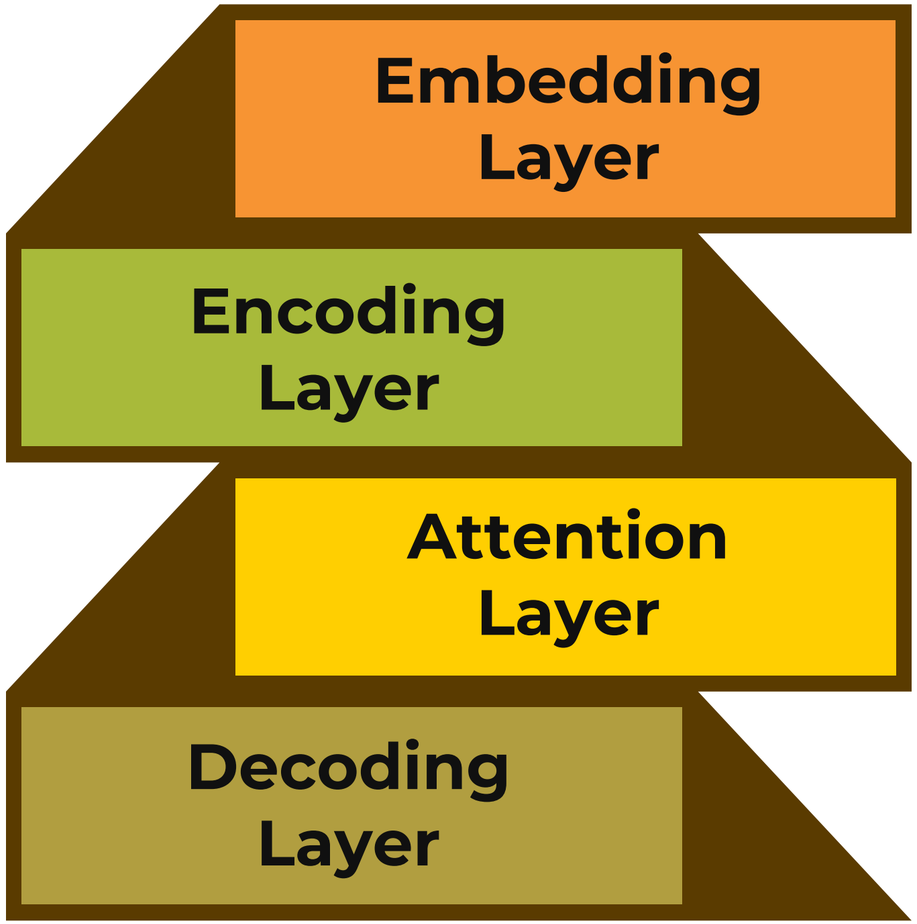
Comparison Works:

- Compared LATTE with BWMP, HAbAD, S-HAbAD, and RepNet.
- Superior performance of LATTE attributed to LSTM depth, self-attention, and OCSVM.

Framework	Memory footprint (KB)	#Model parameters ($x10^3$)	Average inference time (μs)
BWMP [30]	13,147	3435	644.76
HAbAD [34]	4558	64	685.05
S-HAbAD [34]	5600	325	976.65
LATTE	1439	331	193.90
RepNet [28]	5	0.8	68.75

Implementation of Predictor Model

Predictor Model



Implementing the Predictor Model

Embedding Layer:

- **Input Sequences:** The input to the LSTM network is often a sequence of discrete tokens, where each token represents an element in the sequence.
- **Embedding Layer:** The embedding layer takes these integer indices as input and maps each index to a continuous vector representation. This mapping is learned during the training process.

```
self.embedding_layer = nn.Linear(input_size, 128)
```

Implementing the Predictor Model (cont.)

Encoding Layer:

- **Sequential Processing:** LSTM processes input sequences step by step, capturing temporal dependencies.
- **Memory Cell:** LSTM's memory cell selectively stores and retrieves information, aiding in long-term dependency capture.
- **Hidden States:** LSTM's hidden state serves as a compact representation, summarizing the entire input sequence.
- **Layer Stacking:** `***num_layers***` parameter allows stacking for hierarchical feature learning in deeper architectures.
- **Batch Processing:** `**batch_first=True**` indicates input/output tensors as (batch, seq, feature) for batch processing.

In summary, the encoding LSTM efficiently processes sequential data, captures long-term dependencies, and produces a meaningful hidden state for downstream tasks.

```
self.encoder = nn.LSTM(128, hidden_size, num_layers, batch_first=True)
```

Implementing the Predictor Model (cont.)

Attention Layer:

- Enables the model to selectively focus on crucial parts of the input sequence during decoding.
- Attention layer generates the context vector (ϕt) by applying the self-attention mechanism to the encoder outputs
- Attention mechanism begins by applying a linear transformation on the encoder's current state and multiplies the result with the encoder output
- The output from the multiplication is passed through a SoftMax activation to compute the attention weights
- The attention weights are scalars multiplied with the encoder outputs to compute the attention applied vector which is then combined with the encoder output to compute the input to the decoder
- The context vector is given as input to the stacked two-layer decoder

In essence, the attention layer optimizes information processing in LSTM, allowing the model to adaptively focus on key elements for more effective sequence-to-sequence tasks.

```python
class Attn(nn.Module):
    def __init__(self, h_dim):
        super(Attn, self).__init__()
        self.h_dim = h_dim
        self.main = nn.Sequential(
            nn.Linear(h_dim, 24),
            nn.ReLU(True),
            nn.Linear(24, 1)
        )

    def forward(self, encoder_outputs):
        b_size = encoder_outputs.size(0)
        attn_ene = self.main(encoder_outputs.view(-1, self.h_dim)) # (b, s, h) -> (b * s, 1)
        return F.softmax(attn_ene.view(b_size, -1), dim=1).unsqueeze(2) # (b*s, 1) -> (b, s, 1)

class AttnClassifier(nn.Module):
    def __init__(self, h_dim, c_num):
        super(AttnClassifier, self).__init__()
        self.attn = Attn(h_dim)
        self.main = nn.Linear(h_dim, c_num)

    def forward(self, encoder_outputs):
        attns = self.attn(encoder_outputs) #(b, s, 1)
        feats = (encoder_outputs * attns).sum(dim=1) # (b, s, h) -> (b, h)
        return F.log_softmax(self.main(feats)), attns
```

Implementing the Predictor Model (cont.)

Decoding Layer:

- **Context Utilization:** Processes the context vector, summarizing relevant information from the input sequence.
- **Sequential Generation:** Step-by-step generation of the output sequence based on the context and previous predictions.
- **Hidden State Update:** Updates its hidden state at each step, incorporating context and past predictions.
- **Output Prediction:** Predicts the next sequence element using a linear layer applied to the hidden state.
- **Feedback Loop:** Utilizes a feedback loop, feeding its own predictions as input for subsequent steps.
- **Sequence Completion:** Continues decoding until a predefined stopping criterion is met.

In our case decoding layer produces a 64-dimension output that is passed to the last linear layer to obtain a k dimensional output

```
self.decoder = nn.LSTM(128, hidden_size, num_layers, batch_first=True)
```

The Predictor Class

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.model_selection import train_test_split
import torch.nn.functional as F

class Predictor(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers, seq_length):
        super(Predictor, self).__init__()
        self.embedding_layer = nn.Linear(input_size, 128)
        self.encoder = nn.LSTM(128, hidden_size, num_layers, batch_first=True)
        self.attn_classifier = AttnClassifier(hidden_size, output_size)
        self.decoder = nn.LSTM(128, hidden_size, num_layers, batch_first=True)
        self.output_layer = nn.Linear(hidden_size, output_size)
        self.seq_length = seq_length

    def forward(self, x):
        embedded = self.embedding_layer(x)
        encoder_output, (encoder_hidden, _) = self.encoder(embedded)
        attention_output, attns = self.attn_classifier(encoder_output)
        decoder_output, _ = self.decoder(attention_output)
        predicted_output = self.output_layer(decoder_output)

        return predicted_output
```

Training Predictor Model

1. Creating DataLoader from train data

```
# Load CSV data
df = pd.read_csv('train_part.csv')
df = df.fillna(0)

# Assuming you want to predict 'Label', you can split the data into features (X) and labels (y)
X = df[['Signal1', 'Signal2', 'Signal3', 'Signal4']].values
y = df['Label'].values

# Split the 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 NumPy arrays to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

# Create DataLoader for training and testing
batch_size = 64
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, drop_last = True)

test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False, drop_last = True)
```

Training Predictor Model (cont.)

2. Define Hyperparameters and Initialize model

```
# Instantiate the Predictor model
input_size = 4  # Assuming 4 Signal columns in your data
hidden_size = 64
output_size = 4  # Assuming you want to predict a single label
num_layers = 2
seq_length = 10  # Adjust according to your data

predictor_model = Predictor(input_size, hidden_size, output_size, num_layers, seq_length)

# Define loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(predictor_model.parameters(), lr=0.001)
```

Training Predictor Model (cont.)

3. Training Loop

```
# Training loop
num_epochs = 10
for epoch in range(num_epochs):
    predictor_model.train()
    total_loss = 0

    for inputs, labels in train_loader:
        # Forward pass
        outputs= predictor_model(inputs)

        loss = criterion(outputs, inputs)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    print(f'Epoch {epoch+1}/{num_epochs}, Loss: {total_loss/len(train_loader)}')
```

Training Predictor Model (cont.)

4. Testing Loop

```
# Testing loop
predictor_model.eval()
test_loss = 0

with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = predictor_model(inputs)
        test_loss += criterion(outputs, inputs).item()

average_test_loss = test_loss / len(test_loader)
print(f'Average Test Loss: {average_test_loss}')
```

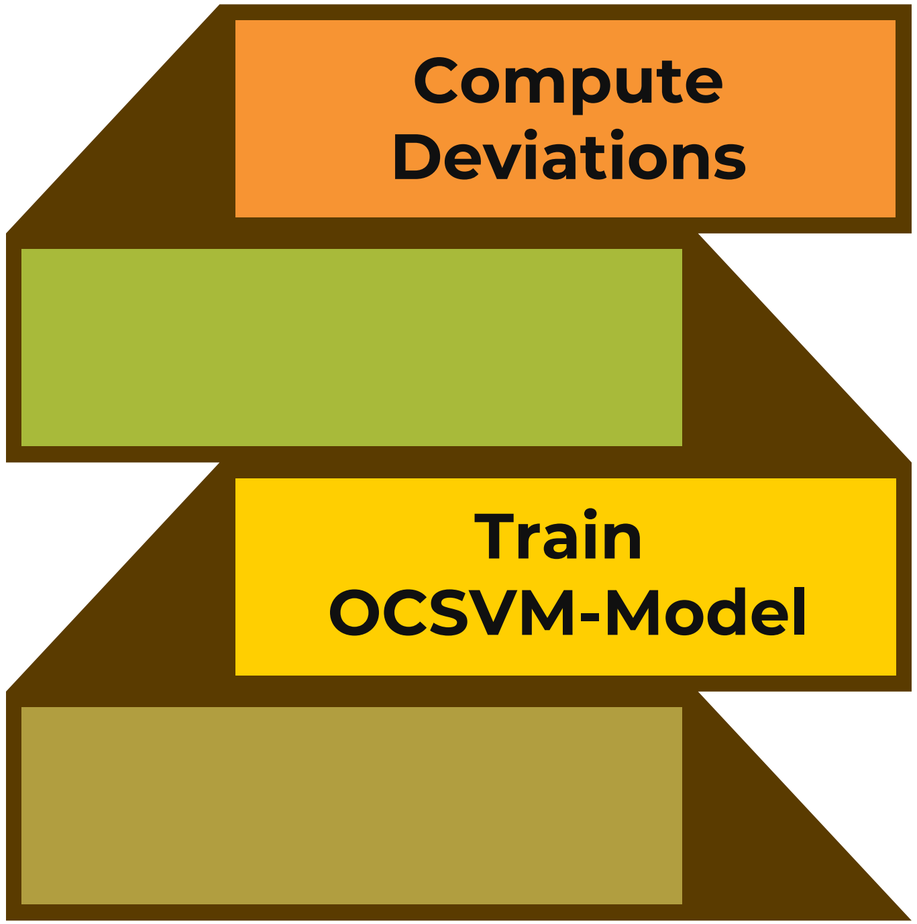
Training Predictor Model (cont.)

- ***Training Results***

```
Epoch 1/10, Loss: 0.16398479044437408
Epoch 2/10, Loss: 0.09632661007344723
Epoch 3/10, Loss: 0.0848440062254667 3
Epoch 4/10, Loss: 0.08514694310724735
Epoch 5/10, Loss: 0.08286971226334572
Epoch 6/10, Loss: 0.08160734362900257
Epoch 7/10, Loss: 0.08301303721964359
Epoch 8/10, Loss: 0.08213127963244915
Epoch 9/10, Loss: 0.08184721320867538
Epoch 10/10, Loss: 0.08211404457688332
Average Test Loss: 0.07942967116832733
```

Detector Model

Detector Model



Detector Model

Compute Deviations:

- The `**compute_deviations**` function (given below) evaluates the LSTM-based predictor model on the data, then calculates the absolute deviations between the model's predictions and true labels, and returns these deviations as a NumPy array.
- This provides a quantitative measure of how well the predictor model is performing on the given dataset, with lower deviations indicating better performance.

```
def compute_deviations(model, data_loader):
    model.eval()
    deviations = []

    with torch.no_grad():
        for inputs, labels in data_loader:
            outputs = model(inputs)
            deviation = torch.abs(outputs.squeeze() - labels)
            deviations.extend(deviation.numpy())

    return np.array(deviations)

# Compute deviations for training and test data
train_deviations = compute_deviations(predictor_model, train_loader).reshape(-1, 1)
# print(train_deviations)
test_deviations = compute_deviations(predictor_model, test_loader).reshape(-1, 1)
```

Detector Model (cont.)

Training the OCSVM-Model:

- Initializes and trains an One-Class SVM (OCSVM) with an RBF kernel and a specified `***nu***` parameter, which controls the trade-off between sensitivity and false positive rate.
- Uses the trained model to predict anomalies in the test deviations. Anomaly predictions are binary `**-1**` for anomalies and `**1**` for normal behavior (however, in our dataset this binary representation was replaced by `**0**` for normal behavior and `**1**` for anomalies, which we have taken care of).
- Ensure proper preprocessing and scaling of the deviations before training and predicting for optimal OCSVM performance.

```
ocsvm_model = OneClassSVM(kernel='rbf', nu=0.1)  # You might need to adjust the hyperparameters
ocsvm_model.fit(scaled_train_deviations)

# Predict anomalies using the trained OCSVM
anomaly_predictions = ocsvm_model.predict(scaled_test_deviations)

# Assuming y_test_tensor is a binary label indicating normal (0) or anomaly (1)
true_labels = y_test_tensor.numpy()

true_labels =  1 - true_labels
```

Detector Model (cont.)

Anomaly Detection Results

- The model achieved the accuracy of 85.94%.

```
# Evaluate the perf. of anomaly detection (you might need to adjust based on your evaluation metric)

accuracy = np.mean(anomaly_predictions == true_labels)
print(f'Anomaly Detection Accuracy: {accuracy * 100:.2f}%')
```

```
Anomaly Detection Accuracy: 85.94%
```

Possible Enhancements

- Implement the LATTE framework on an automotive ECU or simulator to evaluate its real-world effectiveness. The paper evaluates LATTE using a dataset created for a specific purpose, but running it on actual hardware could provide additional and specific insights

- Explore optimizations to reduce LATTE's runtime overhead, such as quantization or pruning of the LSTM model. This could help deploy LATTE on more resource-constrained ECUs.

- Evaluate LATTE's effectiveness at detecting anomalies from sensor failures or faults, in addition to cyberattacks. The paper focuses on security anomalies, but LATTE could potentially detect other types of anomalies.

- Combine LATTE with a specification-based or rule-based technique to improve detection of simple attacks like denial of service. This could help fasten up the process

- Explore ensembles or fusion of LATTE with other anomaly detection techniques like autoencoders or statistical methods. This could improve overall accuracy and robustness.

Code Implementation

https://colab.research.google.com/drive/17fINQJOr8jo1wjaf9Up7wCwU6MHaFFO9#scrollTo=PX7IhGc7kf0I

References

- Research Paper:
  - Vipin Kumar Kukkala, Sooryaa Vignesh Thiruloga, and Sudeep Pasricha. 2021. LATTE: LSTM Self-Att ention based Anomaly Detection in Embedded Automotive Platforms. ACM Trans. Embed. Comput. Syst. 20, 5s, Article 67 (September 2021), 23 page
    - https://dl.acm.org/doi/pdf/10.1145/3476998

- Dataset:
    - https://github.com/etas/SynCAN

Thanks!

(Questions?)