International Journal of Research in Engineering, Science and Management
Volume 8, Issue 8, August 2025
https://www.ijresm.com | ISSN (Online): 2581-5792

Network Attack and Anomaly Detection in IoT Devices

V. S. Chaithra[1], M. S. Vikas[2*], N. S. Sowmya[3], B. Pankaja[4]
[1,3,4]Assistant Professor, Department of CSE-AIML, Sapthagiri NPS University, Bangalore, India
[2]Assistant Professor, Department of CSE-AIML, Sambhram Institute of Technology, Bangalore, India

***Abstract*: In recent years, there has been a significant increase in the use of Internet of Things (IoT) devices in Australia, ranging from simple household appliances like smart furniture and lighting systems to complex machinery and industrial equipment. With the proliferation of IoT, network attacks and anomalies have increasingly come under scrutiny. Especially, the recent network security incidents involving Australian companies have highlighted the importance of attack or anomaly detection. The IoT refers to a network of physical devices, vehicles, and home appliances embedded with sensors, software, and connectivity capabilities, which can collect and exchange data without direct human intervention and communicate through the internet or other networks. With the development of IoT, fog computing has emerged as an important concept, providing computing and storage services at the edge of the network. Fog computing processes data close to the data source, reducing the burden on centralized cloud computing and enhancing the speed and efficiency of data processing. This is particularly applicable to real-time data analysis and processing, playing a significant role in data management and security monitoring within IoT environments.**

***Keywords*: IoT Security, Anomaly Detection, Network Attacks, Edge Computing, Intrusion Detection Systems.**

1. Introduction

In recent years, there has been a significant increase in the use of Internet of Things (IoT) devices in Australia, ranging from simple household appliances like smart furniture and lighting systems to complex machinery and industrial equipment. With the proliferation of IoT, network attacks and anomalies have increasingly come under scrutiny. Especially, the recent network security incidents involving Australian companies have highlighted the importance of attack or anomaly detection. The IoT refers to a network of physical devices, vehicles, and home appliances embedded with sensors, software, and connectivity capabilities, which can collect and exchange data without direct human intervention and communicate through the internet or other networks. With the development of IoT, fog comput- ing has emerged as an important concept, providing computing and storage services at the edge of the net- work. Fog computing processes data close to the data source, reducing the burden on centralized cloud com- puting and enhancing the speed and efficiency of data processing. This is particularly applicable to real-time data analysis and processing, playing a significant role in data management and security monitoring within IoT environments.

2. Problem Statement

IoT devices typically have limited performance, meaning their computational capabilities are restricted or lack the capacity to process data. Therefore, it is necessary to develop models that can run on these limited- performance devices or local routers or servers based on fog computing, to facilitate automatic monitoring of network attacks or anomalies. Our exploratory data analysis (EDA) has found that network attacks and abnormal traffic have periodic occurrences at different times. This finding indicates the importance of time series models for automatic network security monitoring. Currently, most related research and work have not in- tegrated time series models into IoT network security monitoring, whereas their introduction can significantly enhance the robustness of future event predictions.

3. Methodology

A. Proposed Solution

To address the challenge of real-time anomaly detection in resource-constrained IoT environments, we developed a lightweight Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) units. This model was deployed on a Raspberry Pi to evaluate performance in edge settings. Key aspects of the solution include:

- *Efficiency through Quantization and Width Multipliers*: The LSTM model was optimized using quantization and width multipliers, significantly reducing computational load while preserving accuracy.
- *Accurate Anomaly Detection*: The model was trained on the ToN IoT Modbus dataset and achieved reliable detection of network attacks and anomalies.
- *Baseline Comparison*: A lightweight Multilayer Perceptron (MLP) model was used for comparison. The LSTM model consistently outperformed the MLP in both detection performance and generalization.

*Corresponding author: msvikas092@gmail.com

B. Importing and Visualization of Dataset

We used the publicly available ToN IoT_Modbus dataset provided by the University of New South Wales Canberra. It contains IoT telemetry logs in CSV format captured via Modbus protocol. The data includes timestamps, sensor values, and labeled attack types.

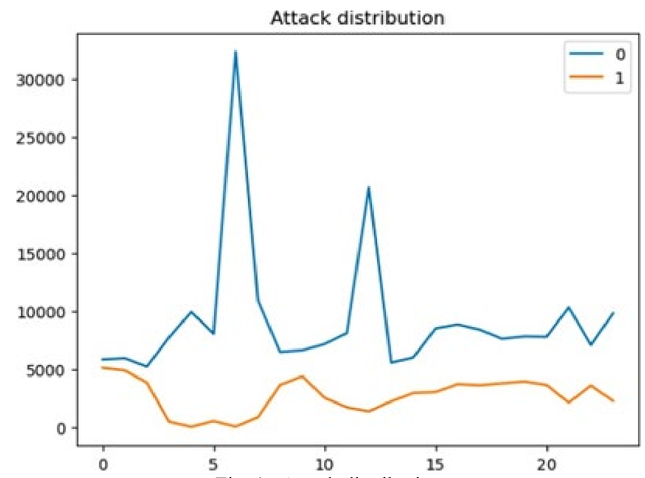

Fig. 1. Attack distribution

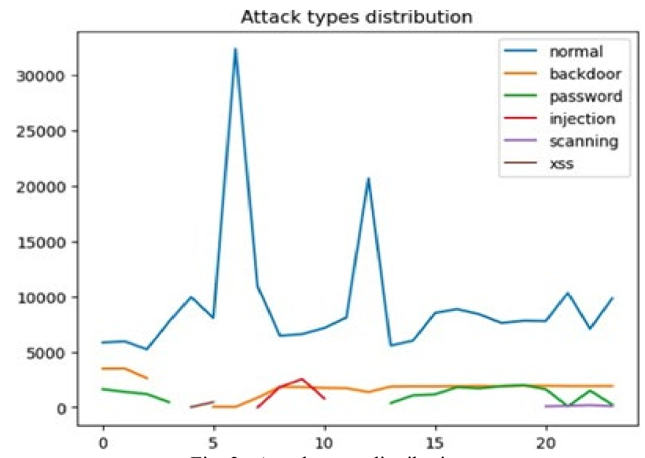

Fig. 2. Attack types distribution

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
data = pd.read_csv('./datasets/IoT_Modbus.csv')
data['datetime'] = pd.to_datetime(data['date'] + ' ' +
        data['time'])
data['hour'] = data['datetime'].dt.hour
label_list = []
for i in range(24):
result = data[data['hour'] == i]['label'].value_counts()
label_list.append([result[0],result[1]])
label_counts =
        pd.DataFrame(columns=['0','1'],data=label_list)
label_counts.plot(title='Attack distribution') type_list =
[]
for i in range(24):
result = data[data['hour'] == i]['type'].value_counts()
type_list.append(result.to_dict())
column_names =
        list(data['type'].value_counts().to_dict().keys())
type_counts =
        pd.DataFrame(columns=column_names,data=type_list)
type_counts.plot(title='Attack types distribution')
```

Listing 1: Importing and preprocessing IoT data

Hourly label distribution and attack types were visualized.

C. Attack Type Visualization

```
type_list = []
for i in range(24):
result = data[data['hour'] == i]['type'].value_counts()
type_list.append(result.to_dict())
column_names =
        list(data['type'].value_counts().to_dict().keys())
type_counts =
        pd.DataFrame(columns=column_names,data=type_list)
type_counts.plot(title='Attack types distribution')
```

Listing 2: Attack type distribution graphs

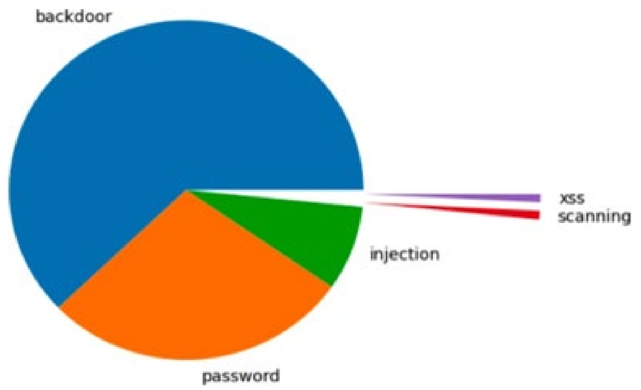

Fig. 3. Attack type pie plot

```
pie_data = data['type'].value_counts().to_dict() explode =
[0,0,0,1,1] plt.pie(x=list(pie_data.values())[1:],
labels=list(pie_data.keys())[1:],explode=explode)
```

Listing 3: Pie plot of attacks

D. Data Preprocessing

Data preprocessing focuses on transforming time series data to enable the LSTM model to learn time dependencies effectively. Date and time fields are cleaned to remove inconsistencies, merged into a unified DateTime format, and used to extract features like year, month, day, hour, and day of the week. The data is sorted chronologically, and the original fields are removed for simplicity. Extracted features are reordered and optimized for efficiency, ensuring the dataset is ready for accurate model processing.

E. Model Training

We trained an LSTM-based recurrent neural network, using width multipliers to adjust the number of hidden units for resource-constrained environments. This approach reduces model complexity while preserving performance, ideal for IoT or edge devices. To address class imbalance, we applied class weights to the loss function, ensuring better recognition of less frequent attack data. The Adam optimizer was used with an appropriate learning rate for stable convergence. We monitored performance metrics like loss, accuracy, precision, recall, and F1 score throughout the training to ensure effective attack detection and resource efficiency.

```
import os
import numpy as np import
pandas as pd import random
from sklearn.metrics import accuracy_score, precision_score,
       recall_score,  f1_score
from sklearn.preprocessing import MinMaxScaler from
sklearn.utils.class_weight import
       compute_class_weight import
torch
from torch import optim import
torch.nn as nn
from torch.utils.data import TensorDataset from
torch.utils.data import DataLoader from matplotlib
import pyplot as plt
from tqdm import tqdm
from datetime import datetime seed =
42 torch.manual_seed(seed)
np.random.seed(seed)
random.seed(seed)
if torch.cuda.is_available():
device = torch.device("cuda")
torch.cuda.manual_seed(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False else:
device = torch.device("cpu")
data = pd.read_csv('datasets/IoT_Modbus.csv')
data['date'] = data['date'].str.strip()
data['time'] = data['time'].str.strip() data['datetime'] =
pd.to_datetime(data['date'] + ' ' +
       data['time'], format='%d-%b-%y %H:%M:%S')
data['year'] = data['datetime'].dt.year data['month'] =
data['datetime'].dt.month data['day'] =
data['datetime'].dt.day data['hour'] =
data['datetime'].dt.hour data['minute'] =
data['datetime'].dt.minute
data['second'] = data['datetime'].dt.second
data['dayofweek'] = data['datetime'].dt.dayofweek # Sort
the data by datetime
data = data.sort_values(by='datetime')
# Drop the original date, time, and timestamp columns
data.drop(['date', 'time', 'datetime', 'type'], axis=1,
        inplace=True)
  # Adjust feature order
order = ['year', 'month', 'day', 'hour', 'minute', 'second',
       'dayofweek', 'FC1_Read_Input_Register',
'FC2_Read_Discrete_Value', 'FC3_Read_Holding_Register',
       'FC4_Read_Coil', 'label']
data = data[order].astype('int32') #
Calculate split points
split_idx = int(len(data) * 0.8)
# Split the data set, keeping order train_data =
data.iloc[:split_idx] test_data =
data.iloc[split_idx:]
# Separate features and labels
X_train = train_data.drop('label', axis=1) y_train
= train_data['label']
X_test = test_data.drop('label', axis=1) y_test =
test_data['label']
feature_columns = [col for col in X_train.columns if col !=
       'label']
scaler = MinMaxScaler()
X_train[feature_columns] =
       scaler.fit_transform(X_train[feature_columns])
.astype('float32')
X_test[feature_columns] =
       scaler.transform(X_test[feature_columns])
.astype('float32')
X_train.info()
```

Listing 4: Cleaning time fields and transformation of extract features

```
class LightweightLSTM(nn.Module):
def __init__(self, input_size, hidden_size, output_size,
       num_layers=1, width_multiplier=1.0):
super(LightweightLSTM, self).__init__()
# Adjust hidden size based on the width multiplier
adjusted_hidden_size = int(hidden_size *
       width_multiplier)
# Define the LSTM
layer
self.lstm = nn.LSTM(input_size, adjusted_hidden_size,
       num_layers=num_layers, batch_first=True)
self.fc = nn.Linear(adjusted_hidden_size, output_size) def
forward(self, x):
# LSTM layer
lstm_out, _ = self.lstm(x)
# Take the output of the last time step
last_time_step_out = lstm_out[:, -1, :] #
Output layer
out = self.fc(last_time_step_out)
return out
features_num = X_train.shape[1]
hidden_neurons_num = 512
output_neurons_num = 1
lstm_num_layers = 2
multiplier = 0.5
model = LightweightLSTM(features_num,
       hidden_neurons_num, output_neurons_num,
       lstm_num_layers,
multiplier).to(device)
class_weights =
       compute_class_weight(class_weight='balanced',
       classes=np.unique(y_train), y=y_train)
class_weights = torch.tensor(class_weights,
       dtype=torch.float).to(device=device)
weights = torch.tensor([1, class_weights[1]], dtype=torch.float)
criterion = nn.BCEWithLogitsLoss(torch.FloatTensor
       ([weights[1] / weights[0]])).to(device)
optimizer = optim.Adam(model.parameters(), lr=0.0005) batch_size
= 128
X_train_tensor = torch.tensor(X_train.values).float().unsqueeze(1)
.to(device)
y_train_tensor =
       torch.tensor(y_train.values).float().unsqueeze(1). to(device)
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset,
batch_size=batch_size, shuffle=False) Training
loop:
num_epochs = 100
pbar = tqdm(total=num_epochs)
loss_list = [None] * num_epochs
acc_list = [None] * num_epochs for
epoch in range(num_epochs):
model.train()
running_loss = 0.0
running_accuracy = 0.0
times = 0
for inputs, labels in train_loader:
# FP
outputs = model(inputs)
loss = criterion(outputs, labels) #
BP and optimization
optimizer.zero_grad() loss.backward()
optimizer.step()
# Calculate indicators
model.eval()
with torch.no_grad():
probabilities = torch.sigmoid(outputs)
predictions = (probabilities >
       0.5).float().cpu().numpy()
# Calculate indicators
y = labels.cpu().numpy()
running_loss += loss.item() * inputs.size(0)
running_accuracy += accuracy_score(y, predictions) times +=
1
epoch_loss = running_loss / len(train_loader.dataset) accuracy =
running_accuracy / times
loss_list[epoch] = epoch_loss
acc_list[epoch] = accuracy
print(f'Epoch [{epoch+1}/{num_epochs}], Loss:
       {epoch_loss}, Accuracy: {accuracy}')
pbar.update(1)
pbar.reset()
```

Listing 5: LSTM model using Adam optimizer

F. Plotting Loss and Accuracy

```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(loss_list, label='Training Loss')
plt.title('Training Loss per Epoch')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
# Draw accuracy curve
plt.subplot(1, 2, 2)
plt.plot(acc_list, label='Training Accuracy') plt.title('Training
Accuracy per Epoch') plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend() plt.show()
X_test_tensor    =    torch.tensor(X_test.values).float()
.unsqueeze(1).to(device) model.eval()
outputs = model(X_test_tensor) with
torch.no_grad():
probabilities = torch.sigmoid(outputs) predictions
= (probabilities >
        0.5).float().cpu().numpy() #
Calculate indicators
acc = accuracy_score(y_test, predictions) precision =
precision_score(y_test, predictions) recall =
recall_score(y_test, predictions)
f1 = f1_score(y_test, predictions)
print("Accuracy: ", acc, ", Precision: ", precision, ", Recall:
        ", recall, ", F1: ", f1)
```

Listing 6: Plotting Loss and Accuracy

G. Model Saving

```
save_folder = "save_model"
if not os.path.exists(save_folder):
os.makedirs(save_folder) current_time =
        datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
model_filename = f"model_lstm_{current_time}.pt" if
multiplier == 1:
model_filename = f"model_lstm_{current_time}
_without_width_multiplier.pt"
full_path = os.path.join(save_folder, model_filename)
torch.save(model.state_dict(), full_path) print("Model saved
as:", full_ath)
```

Listing 7: Saving the model

4. Results and Discussion

To ensure the fairness of our experiments, we used the same training and test datasets. Notably, the time features in the test dataset follow those in the training dataset, which was done to validate the model's effectiveness in detecting future data. Additionally, to test the model performance in an actual resource- constrained environment, we configured the Raspberry Pi to use only one core. This is because IoT devices often need to reserve computing resources for their own services, and we also wanted to assess how the inference devices would perform under extreme pressure, i.e., the extent of network data stress they can handle.

A. Comparison Between RNN and Baseline Model

Our analysis showed that the baseline Multilayer Perceptron (MLP) model underperformed compared to the Recurrent Neural Network (RNN) with Long Short- Term Memory (LSTM) units. In our task, we focused on the model's ability to classify positive samples (i.e., attacks or anomalies). The MLP model's recall rate was only 0.322927, resulting in a low F1 score of 0.47614, indicating its ineffectiveness in identifying positive samples. In contrast, the RNN model achieved an accuracy of 0.932798 and a recall rate of 0.639914, demonstrating superior performance in detecting future data.

- Accuracy: 0.9328
- Precision: 0.8336
- Recall: 0.6399
- F1 Score: 0.7240

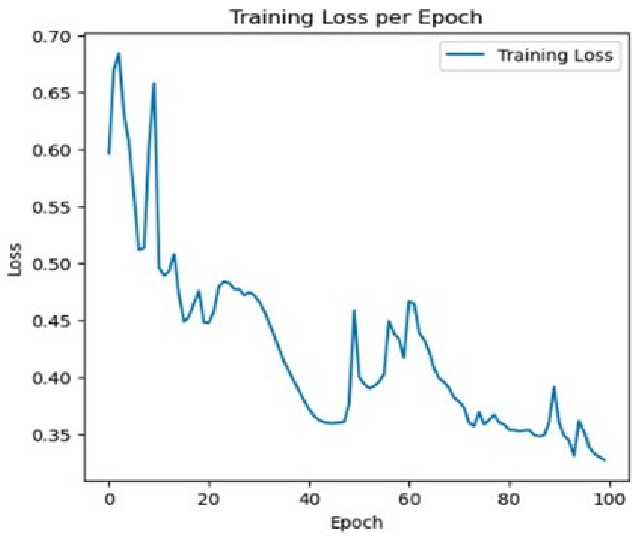

Fig. 4. Training loss per epoch

B. Impact of Width Multipliers and Quantization

- In the scenario without width multipliers and quantization, the model was the most resource-intensive. In this case, the model took 26 minutes and 53 seconds to infer 57,439 samples, processing about 36 network traffic data per second. This inference speed is clearly insufficient for models deployed on local servers or routers.

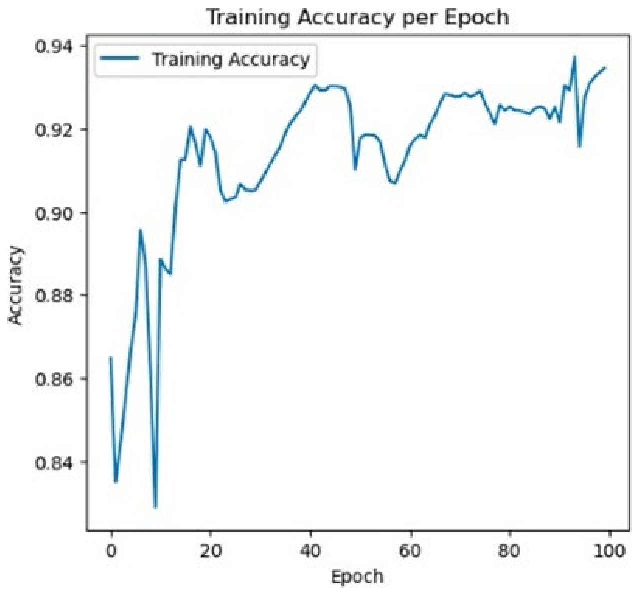

Fig. 5. Training accuracy per epoch

- The inference speed of the model using width multipliers was significantly improved, requiring only 8 minutes and 50 seconds to complete all samples, processing about 108 network traffic data per second. Moreover, the model performed best in this test, validating the effectiveness of width multipliers in enhancing inference speed on constrained devices.

- The quantized model's inference speed was even faster, taking only 7 minutes and 22 seconds to process all network traffic data, which equals processing about 130 data per second. This demonstrates that model quantization can significantly accelerate the inference process.
- When combining width multipliers and model quantization, the model's performance was impressively efficient. It took only 3 minutes and 25 seconds to infer all test dataset samples, processing about 279 data per second. Although there was a slight performance drop (about 0.1%), the model still retained fairly good performance. Compared to the initial model, which processed 36 data per second, this improvement to 279 data per second is substantial.

5. Conclusion

In this project, we developed an IoT anomaly detection system leveraging RNN models with LSTM units to ensure real-time and efficient identification of net- work anomalies. By optimizing for resource-constrained environments, we implemented quantization and width multipliers to reduce computational overhead. The system effectively processes time-series data, achieving robust detection of network attacks. Through integration with fog computing, we decentralized anomaly detection to enhance performance and scalability. Our approach demonstrates significant improvements in accuracy and efficiency, making it suitable for IoT networks with limited resources.

References

[1] Althubiti, S., & Jones, D. M. (2018). LSTM for Anomaly-Based Network Intrusion Detection. Semantic Scholar.

[2] Abirami, S., & Santhi, B. (2020). Network Traffic Anomaly Detection Using LSTM-Based Autoencoder.

[3] Yin, C., Zhu, Y., Fei, J., & He, X. (2017). Anomaly Detection in Cyber Security Attacks on Networks Using MLP Deep Learning. *IEEE Transactions on Information Forensics and Security*.

[4] Radford, S., Dey, S., & Chakraborty, A. (2018). Network Traffic Anomaly Detection Using Recurrent Neural Networks.

[5] Chen, J., Chen, X., & Hu, Y. (2021). A Deep Learning Ensemble for Network Anomaly and Cyber-Attack Detection. *Computers, Materials & Continua*.

[6] Shamshirband, S., Chronopoulos, A. T., & Ghaffari, M. (2021). The Impact of Artificial Neural Network Architecture on Network Attack Detection. *ACM Digital Library*.

[7] Dhaliwal, A., & Khan, F. (2020). Anomaly-Based Intrusion Detection System: A Deep Learning Approach. NSF Public Access Repository.