PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
الجمهـوريـــة الجـزائـريـــة الديـمـقـراطيـــة الشـعـبيـــة
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH
وزارة التعليـم العالـي والبحـث العلمـي
ENGINEERING SCHOOL OF COMPUTER SCIENCE AND DIGITAL TECHNOLOGIES-BEJAIA
المدرسة العـــليا في علـــوم وتكنولوجيات الإعـــلام الآلـــي بجـــاية

Dissertation Submitted to the Department Of Computer Science in Partial Fulfillment of the Requirements for Engineer's Degree in Computer Science

Specialty: Artificial Intelligence and Data Sciences

Submitted By:

Ms. *BOUTAOUI Nada*

Energy Efficiency In Wireless Sensor Networks

Supervised by:

Pr. *BAGAA Miloud - UQTR*

Pr. *AZOUAOU Faiçal - ESTIN*

Members of jury:

▪ **Dr.** AMROUNI Samia	President	MAB/ESTIN
▪ **Dr.** CHELOUAH Leila	Examiner	MCB/ESTIN
▪ **Dr.** KACI Amina	Examiner	MAB/ESTIN
▪ **Mr.** Yanis HAMMOUM	Examiner	DR/ESTIN

Academic year: 2023/2024

Dedication

I would like to dedicate this modest work to my family and friends, who have always supported and guided me through every phase of my life.

To my beloved parents, ***Houria*** *and* ***Hocine****, I can never thank you enough for your unconditional love, support, and encouragement. You have always been my role models and always will be. Thank you for always believing in me, praying for my success and protection. May Allah grant you the highest place in His Jannah.*

To my siblings, ***Houssam Eddine*** *and* ***Nouha****: Thank you for always listening to me and helping me believe in myself. Your patience with your little sister (me, XD) and the safe space you provided have meant the world to me. I wish you nothing but the greatest success, Insha'Allah.*

To my dear aunt, ***Habiba****, Your kind heart has always been a source of inspiration for me. Your words of encouragement and prayers have helped me every step of the way.*

To my roommate, classmate, and best friend, ***Ikram****, Thank you so much for putting up with me and being so understanding. I'm so lucky to have you.*

To all of my high school friends, who have stood by me for these past eight years, cheering me up along the way,your support means more to me than you can imagine. And to my university friends, who made my university experience one of the best and most cherished times of my life, thank you for making life funnier and more beautiful.

But Especially, to our brothers and sisters in Gaza. This year, you have taught me lessons that no higher education could ever provide. You have shown me faith, resilience, strength, and the meaning of complete submission to Allah. I cannot thank you enough for offering me a new perspective on life. May Allah be with you.

Finally, I would like to thank anyone I have not mentioned who has helped and taught me, shaping me into the person I am today.

Nada Boutaoui

Acknowledgements

Bismiallah And Alhamdoulilah, All Praise and Thanks are due to Allah the most merciful and the most compassionate for helping me and providing me with the opportunity and the strength to complete this humble work, may Allah use us for what is best for us and for the world.

I would like to to express my gratitude to my project supervisors Pr. Miloud Bagaa, Pr. Azouaou Faiçal, PHD Student Saif Eddine Khelifa who offered me the opportunity to work on this project, their expertise, guidance and especially patience and understanding throughout this project period.

I would like to thank the members of the jury for their interest and for taking the time to evaluate this work.

I would also like to acknowledge the teachers and staff of our university: l'École nationale Supérieure des sciences et technologies de l'informatique et du numérique.

Thank you

Abstract

Wireless Sensor Networks (WSNs) face significant challenges in terms of energy efficiency, requiring strategies that balance power consumption, data quality, and network reliability to maximize network lifetime. This issue becomes more complex with the growing adoption of IoT applications and the dynamic nature of sensor deployment environments. In this context, energy-efficient solutions for WSNs have gained prominence, especially those incorporating machine learning and graph-based techniques.

This thesis investigates energy optimization strategies for Wireless Sensor Networks (WSNs) in the context of IoT applications and dynamic deployment environments. We introduce Delta-GCN, a novel model that combines Graph Convolutional Networks (GCNs) with a customized Long Short-Term Memory (LSTM) layer to capture spatial and temporal dependencies in sensor networks. The model is integrated into a Federated Learning (FL) framework, enabling decentralized training while preserving data privacy.

Our semi-supervised approach addresses the challenge of missing data in WSNs through data imputation, simultaneously optimizing energy consumption by reducing unnecessary sensor activity. Extensive empirical evaluations demonstrate that Delta-GCN significantly outperforms existing state-of-the-art methods across key performance metrics, including F2 score, precision, recall, energy consumption, and network lifetime. The research findings underscore the potential of integrating GCNs and FL for developing scalable, energy-efficient WSN deployments. This work contributes to the field by advancing energy management strategies applicable to real-world sensor networks, paving the way for more sustainable and efficient IoT infrastructures.

Keywords: Wireless Sensor Networks, Energy Optimization, Graph Convolutional Networks, Federated Learning, IoT

ملخص

تواجه شبكات الاستشعار اللاسلكية (WSNs) تحديات كبيرة فيما يتعلق بالكفاءة الطاقوية، حيث تتطلب استراتيجيات توازن بين استهلاك الطاقة وجودة البيانات وموثوقية الشبكة بهدف إطالة عمر الشبكة إلى أقصى حد. تتعقد هذه المشكلة مع التزايد المستمر لاعتماد تطبيقات إنترنت الأشياء (IoT) والطبيعة الديناميكية لبيئات نشر أجهزة الاستشعار. في هذا السياق، اكتسبت الحلول الموفرة للطاقة في شبكات الاستشعار اللاسلكية أهمية متزايدة، خاصة تلك التي تدمج تقنيات التعلم الآلي والأساليب المعتمدة على الرسوم البيانية.

تبحث هذه الأطروحة في استراتيجيات تحسين الطاقة لشبكات الاستشعار اللاسلكية (WSNs) في سياق تطبيقات إنترنت الأشياء (IoT) وبيئات النشر الديناميكية. نقدم نموذج "دلتا-جي سي إن" (Delta-GCN)، وهو نموذج جديد يجمع بين الشبكات العصبية التلافيفية الرسومية (GCNs) وطبقة ذاكرة طويلة قصيرة الأمد المخصصة (LSTM) لالتقاط التبعيات المكانية والزمانية في شبكات الاستشعار. يتم دمج النموذج في إطار التعلم الفيدرالي (FL)، مما يتيح التدريب اللامركزي مع الحفاظ على خصوصية البيانات.

يعمل منهجنا شبه الإشرافي على معالجة تحدي فقدان البيانات في شبكات الاستشعار اللاسلكية من خلال استكمال البيانات المفقودة، مع تحسين استهلاك الطاقة عن طريق تقليل النشاط غير الضروري لأجهزة الاستشعار. أظهرت التقييمات التجريبية المكثفة أن نموذج "دلتا-جي سي إن" يتفوق بشكل ملحوظ على الأساليب الحالية في العديد من المقاييس الأساسية، بما في ذلك درجة 2F، والدقة، والاسترجاع، واستهلاك الطاقة، وعمر الشبكة.

تؤكد نتائج البحث على إمكانات دمج الشبكات العصبية التلافيفية الرسومية والتعلم الفيدرالي لتطوير نشرات شبكية لاسلكية مستدامة وموفرة للطاقة. تسهم هذه الدراسة في مجال إدارة الطاقة من خلال تقديم استراتيجيات قابلة للتطبيق على شبكات الاستشعار في العالم الحقيقي، مما يمهد الطريق لبنى تحتية أكثر استدامة وكفاءة لإنترنت الأشياء.

الكلمات المفتاحية: شبكات الاستشعار اللاسلكية، تحسين الطاقة، الشبكات العصبية التلافيفية الرسومية، التعلم الفيدرالي، إنترنت الأشياء

Résumé

Les réseaux de capteurs sans fil (WSNs) font face à des défis importants en matière d'efficacité énergétique, nécessitant des stratégies qui équilibrent la consommation d'énergie, la qualité des données et la fiabilité du réseau afin de maximiser la durée de vie du réseau. Ce problème devient plus complexe avec l'adoption croissante des applications IoT et la nature dynamique des environnements de déploiement des capteurs. Dans ce contexte, les solutions économes en énergie pour les WSNs gagnent en importance, en particulier celles qui intègrent des techniques d'apprentissage automatique et basées sur les graphes.

Cette thèse examine les stratégies d'optimisation énergétique pour les réseaux de capteurs sans fil (WSNs) dans le cadre des applications IoT et des environnements de déploiement dynamiques. Nous introduisons Delta-GCN, un modèle novateur qui combine les réseaux convolutionnels de graphes (GCNs) avec une couche Long Short-Term Memory (LSTM) personnalisée pour capturer les dépendances spatiales et temporelles dans les réseaux de capteurs. Le modèle est intégré dans un cadre d'apprentissage fédéré (FL), permettant un entraînement décentralisé tout en préservant la confidentialité des données.

Notre approche semi-supervisée aborde le problème des données manquantes dans les WSNs via l'imputation de données, tout en optimisant la consommation d'énergie en réduisant l'activité inutile des capteurs. Des évaluations empiriques approfondies montrent que Delta-GCN dépasse largement les méthodes existantes à l'état de l'art sur des indicateurs de performance clés, y compris le score F2, la précision, le rappel, la consommation d'énergie et la durée de vie du réseau.

Mots-clés : Réseaux de capteurs sans fil, Optimisation énergétique, Réseaux convolutionnels de graphes, Apprentissage fédéré, IoT

Abbreviations

AWGN: **A**dditive **W**hite **G**aussian **N**oise

CNN: **C**onvolutional **N**eural **N**etwork

CRPS: **C**ontinuous **R**anked **P**robability **S**core

DPM: **D**iffusion **P**robabilistic **M**odel

DP: **D**ifferential **P**rivacy

DRL: **D**eep **R**einforcement **L**earning

DQN: **D**eep **Q**-**N**etworks

EC: **E**vent **C**orrelations

FedAvg: **F**ederated **A**veraging

FedDyn: **F**ederated **D**ynamic **A**veraging

FedProx: **F**ederated **P**roximal

FL: **F**ederated **L**earning

GAN: **G**enerative **A**dversarial **N**etwork

GAT: **G**raph **A**ttention **N**etwork

GCN: **G**raph **C**onvolutional **N**etwork

GNN: **G**raph **N**eural **N**etwork

GSAVES: **G**raph **S**ensor **A**d**V**ersarial for **E**nergy **S**aving

HMM: **H**idden **M**arkov **M**odel

JGD: **J**oint **G**aussian **D**istribution

LDP: **L**ocal **D**ifferential **P**rivacy

LSTM: **L**ong **S**hort-**T**erm **M**emory

MAC: **M**edia **A**ccess **C**ontrol

MAE: **M**ean **A**bsolute **E**rror

MARL: **M**ulti-agent **R**einforcement **L**earning

MDP: **M**arkov **D**ecision **P**rocess

MITes: **M**IT **E**nvironmental **S**ensors

ML: **M**achine **L**earning

NN: **N**eural **N**etwork

PCA: **P**rincipal **C**omponent **A**nalysis

PIR: **P**assive **I**nfrared

ReLU: **R**ectified **L**inear **U**nit

RL: **R**einforcement **L**earning

RMSE: **R**oot **M**ean **S**quare **E**rror

RNN: **R**ecurrent **N**eural **N**etwork

SAML: **S**elf-**A**dapting **M**AC **L**ayer

SARSA: **S**tate–**A**ction–**R**eward–**S**tate–**A**ction

SCAFFOLD: **S**tochastic **C**ontrolled **A**veraging for **F**ederated **L**earning

SVM: **S**upport **V**ector **M**achine

TARNet: **T**ask-**A**ware **R**econstruction for **T**ime-Series **T**ransformer

TNC: **T**ime **N**eural **C**onnections

TS-TCC: **T**ime **S**eries - **T**ransformation and **C**ross-**C**orrelation

TST: **T**ime **S**eries **T**ransformer

WSN: **W**ireless **S**ensor **N**etwork

List of Figures

Contents

Part I

State Of The Art

Chapter 1

Wireless Sensor Networks and Energy Efficiency

This chapter reviews energy efficiency in WSNs, explaining the need for effective energy management due to the power limitations of sensor nodes. It begins by discussing the main factors contributing to energy consumption, such as transmission power and communication overhead. Next, it explores traditional energy-saving approaches, including duty cycling, data aggregation, and energy harvesting.

The chapter also introduces modern techniques, particularly the use of ML, to optimize energy usage and extend network lifetime. The goal is to provide a clear understanding of how these methods address the challenge of prolonging the operational life of WSNs.

1.1 Basic Concepts

1.1.1 Overview of Wireless Sensor Networks

Wireless Sensor Networks (WSNs) are distributed systems of sensor nodes, each node is a small and autonomous device that has a microprocessor, a wireless communication module and it also comes with various sensors(temperature, motion, cameras..). These nodes perform the task of collectively gathering information from their environment, then they transmit this data to a central gateway through a multi-hop communication, the data is then forwarded to a central processing unit for further analysis and decision making. Figure 1.1 illustrates the structure of a usual WSN

We can model the Energy consumption E of each sensor node as:

$$E = E_{\text{Tx}} + E_{\text{Rx}} + E_{\text{comp}}, \tag{1.1.1}$$

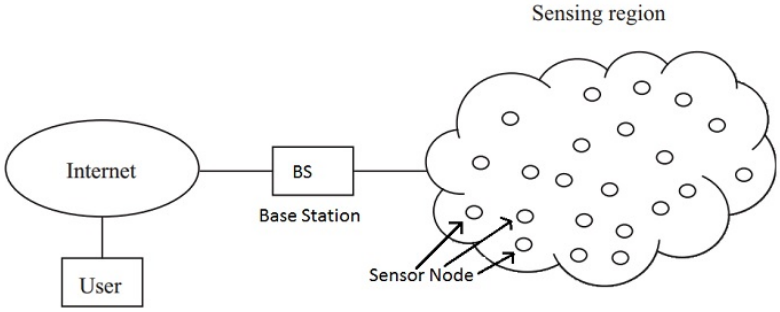


Figure 1.1: General architecture of a Wireless Sensor Network

where E_{Tx} is the energy consumed during transmission, E_{comp} during computation, and E_{Rx} during reception .

Transmission power, distance between nodes, and data processing requirements are the main factors that affect the energy consumption of each node.

1.1.2 Importance of Energy Efficiency in WSNs

Due to the limited battery life of sensor nodes, energy conservation became a crucial concern in WSNs, especially that sensor nodes are usually found in remote and inaccessible areas that makes the replacement of batteries impractical and time consuming, most of the battery life goes to communication processes since the energy required to transmit a bit of data is much higher than one used for computation. The importance of energy efficiency is evident in the fact that the operational life of the network can be extended by optimizing energy usage during the sensing, data processing and communication phases.

Figure 1.2 illustrates the energy consumption breakdown in a typical WSN, showing the need for energy-efficient communication protocols.

1.1.3 Key Challenges in Energy Management

Several challenges are encountered when managing energy consumption in WSNs, including computational costs, communication overhead and the trade off between the network performance and energy saving, the high frequency of data transmission and reception make the communication overhead the primary source of energy drain in WSNs. To reduce the amount of data transmitted, several techniques emerged such as data aggregation, compression and clustering.

Another major challenge is the trade-off between the network performance and the energy conservation, if we reduce the transmission power we can save more energy but on the other hand we may result in a decreased communication range and a higher latency

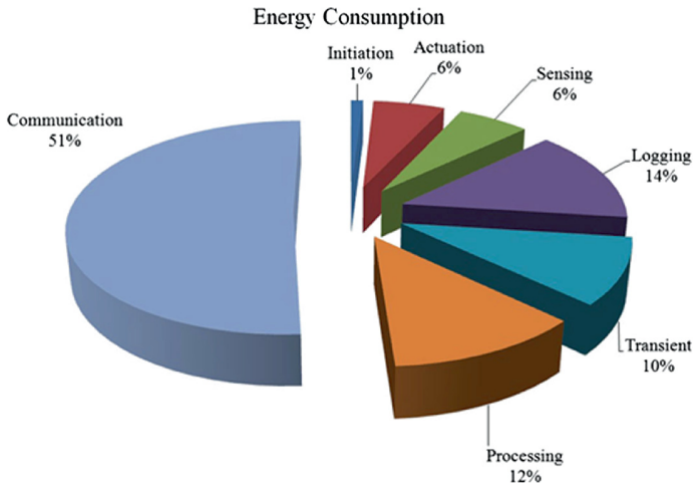


Figure 1.2: Energy Consumption Breakdown in a typical WSN

costing us the network reliability and data delivery rates.

Computational costs are another critical challenge, including the energy used in algorithm execution and data processing, to effectively manage those costs and to prevent premature depletion of node batteries, which could lead to network partitioning and reduced data reliability it is essential to use advanced routing protocols that balance energy consumption across the nodes network.

1.2 Traditional Energy-Saving Techniques:

In Figure 1.3 we can see the breakdown of the different methods that can be used to save energy in WSNs, each focusing on a different perspective, we will discuss some of the main strategies further in this section.

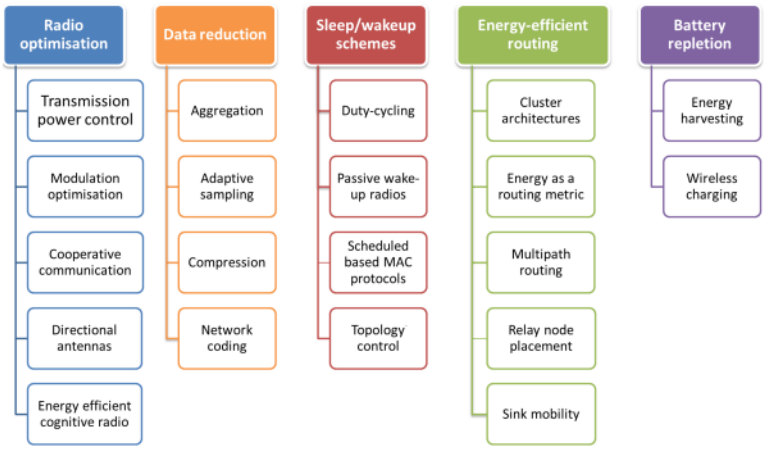


Figure 1.3: Energy saving techniques in WSNs

1.2.1 Duty Cycling

One of the fundamental energy-saving mechanisms in WSNs is Duty cycling, where sensor nodes are periodically switching between the active and sleep states, this method enables the prolongation of the useful life of a network of small battery-powered devices (sensor nodes), allowing those energy constrained devices to monitor physical or environmental conditions for an extended period of time.

Duty cycling addresses this challenge by reducing the amount of time nodes spend in high-powered activity modes. Essentially, nodes alternate between short periods of activity and longer periods of low-power sleep. This approach reduces power consumption significantly, as nodes consume much less power when they are in sleep mode.

We can define the duty cycle as follows:

$$\text{Duty Cycle} = \frac{T_{\text{active}}}{T_{\text{active}} + T_{\text{sleep}}} \tag{1.2.1}$$

where T_{active} is the active time and T_{sleep} is the sleep time.

Duty cycling balances between energy conservation and network performance. When the duty cycle decreases the energy consumption is also reduced, but that costs the network a potential data loss and an increased latency. We can express this relationship as follows:

$$E_{\text{total}} \propto \text{Duty Cycle} \tag{1.2.2}$$

$$\text{Latency} \propto \frac{1}{\text{Duty Cycle}} \tag{1.2.3}$$

where E_{total} represents the total energy consumption. For example, if we consider a sensor node that consumes 50 mW when in active state and 1 mW when sleeping. With a 10 percent duty cycle, its average power consumption is going to be:

$$P_{\text{avg}} = (50\ \text{mW} \times 0.1) + (1\ \text{mW} \times 0.9) = 5.9\ \text{mW} \tag{1.2.4}$$

This represents a great reduction compared to the 50 mW that would be consumed in constant active state. Researchers have proposed a variety duty cycling protocols to effectively implement this technique:

1. Synchronous protocols (e.g: S-MAC, T-MAC): in this case, the sleep and awake periods are synchronized between neighboring nodes ensuring communication by making them awake simultaneously. For example the time is divided into fixed length frames in Sensor-MAC, starting with a synchronization period then a data transmission period.

2. Asynchronous protocols (e.g: B-MAC, X-MAC): Here nodes are allowed to have independent sleep schedules, communication is usually ensured using techniques like preamble sampling.

3. Hybrid protocols (e.g:Z-MAC): These protocols combine features of both synchronous and asynchronous approaches to adapt to varying network conditions. Z-MAC, for instance, behaves like CSMA under low contention and TDMA under high contention.

1.2.2 Data Aggregation and Compression

Data aggregation and compression are two fundamental techniques to save energy in WSNs by reducing the amount of data exchanged over the network. These techniques exploit spatial and temporal correlation of sensory information in reducing redundancy and optimizing communication .

Aggregation is the process of collecting data from multiple sensors or multiple readings by one sensor to minimize the overall amount of data transmitted. We can achieve this with very good results in WSNs because of the natural correlation between the data gathered by sensors in proximity of each other or readings at consecutive times. Data aggregation can be represented as:

$$D_{\text{agg}} = F(D_1, D_2, ..., D_n) \tag{1.2.5}$$

where D_{agg} is the aggregated data, F is the aggregation function, and $D_1, D_2, ..., D_n$ are the individual sensor readings.

On the other hand, data compression can be divided into 2 methods:

1. Lossless compression: Here the original data is perfectly reconstructed with techniques like Huffman coding where shorter codes are assigned to frequent symbols or like Lempel-Ziv-Welch which Builds a dictionary of repeated patterns.

2. Lossy compression: Here we can achieve a higher compression rate but with some loss in information, techniques that follow this method include Transform coding which Applies transforms like DCT (Discrete Cosine Transform) to represent data more compactly or Predictive coding that uses predictions based on previous values to encode the differences

1.2.3 Energy Harvesting:

Energy harvesting is a technique that allows sensor nodes in WSNs to collect energy from surrounding sources int the WSNs' environment, this provides renewable energy sources to

recharge the sensors' battery allowing an extended lifetime of the WSN.

There are various sources used to harvest energy from in WSNs, like solar energy, a very common technique for outdoor WSNs, Thermal energy, Vibration energy and even Radio Frequency energy.

The energy balance of a harvesting-enabled sensor node can be seen as:

$$E_{\text{available}}(t) = E_{\text{initial}} + \int_0^t (P_{\text{harvested}}(\tau) - P_{\text{consumed}}(\tau))d\tau \tag{1.2.6}$$

where $E_{\text{available}}(t)$ is the available energy at time t, E_{initial} is the initial energy, $P_{\text{harvested}}(\tau)$ is the harvested power, and $P_{\text{consumed}}(\tau)$ is the power consumed by the node.

On the other hand, Energy-neutral operation is a key goal in energy harvesting WSNs, and it can achieved when:

$$\lim_{t\to\infty} \frac{1}{t} \int_0^t P_{\text{harvested}}(\tau)d\tau \geq \lim_{t\to\infty} \frac{1}{t} \int_0^t P_{\text{consumed}}(\tau)d\tau \tag{1.2.7}$$

This condition ensures that, on average, the harvested energy is sufficient to power the node's operations indefinitely.

By combining various techniques like duty cycling and energy harvesting we can extend the lifetime of WSNs and save energy, but that is not simple and it requires many design and hardware adjustments, this presents a challenge for other fields like mechanical and electrical engineering.

1.3 Modern Approaches for Energy Saving in WSNs:

1.3.1 Introduction to Machine Learning in WSNs:

Machine learning is a powerful technique that has a large variety of algorithms that can perform different complex tasks in an optimal way after learning from the given data, and within the IoT ecosystem and especially in WSNs, we can find different ways to save energy by addressing different process and optimizing, and for that we can choose suitable ML approaches to analyze complex patterns in the network traffic, sensory data, and the overall energy consumption to make smart decisions that optimize the energy effeciency.

ML is applied for different tasks in WSNs including:

- Predictive maintenance, where the ML model predicts where maintenance is needed before the failure of sensors.

- Adaptive duty cycling, here the model can predict the optimal way of switching sensors on and off to conserve energy.
- Smart Routing, ML can choose the best paths for data transmission, this minimises the communication process.
- Anomaly detection: By identifying unusual behavior (like a malfunctioning sensor), the model can act fast to prevent bigger energy losses.

These applications are deployed using different ML models, including supervised, unsupervised and semi-supervised learning, in the following section we will give more details about how every model works and what problematic is addresses.

1.3.2 Supervised Learning Approaches:

In supervised learning the model is trained on a labeled dataset (inputs: features, and known outputs: target) , the model learns the relationships between the inputs and the outputs, to use it later to predict unknown output. Supervised learning is used a lot to solve problems in the context of WSNs, like, localization and objects targeting, event detection and query processing, media access control , security and intrusion detection... Common ML algorithms include:

Decision Trees and Random Forests

Decision trees are used to handle non-linear datasets, and it is used for classification problems. The input data is iterated through a learning tree to predict the label. Random forests are an extension of decision trees where multiple decision trees are aggregated to improve the robustness and generalization. **Application:** Optimizing cluster head selection in hierarchical WSNs by considering factors like residual energy, node density, and distance to the base station.

Figure 1.4 shows the decision tree classifier used to select the optimal MAC algorithm in an SAML architecture proposed by Sha et al. in “Self-Adapting MAC Layer” .

Support Vector Machines (SVMs)

SVMs is a classification model that learns to find the optimal hyperplane or kernel that separates the classes in the feature space, maximizing the gap between different classes, and new input will be classified based on which side of the gaps are on

Application: Classifying network traffic patterns to predict congestion and adjust transmission power, other application include security and localization.

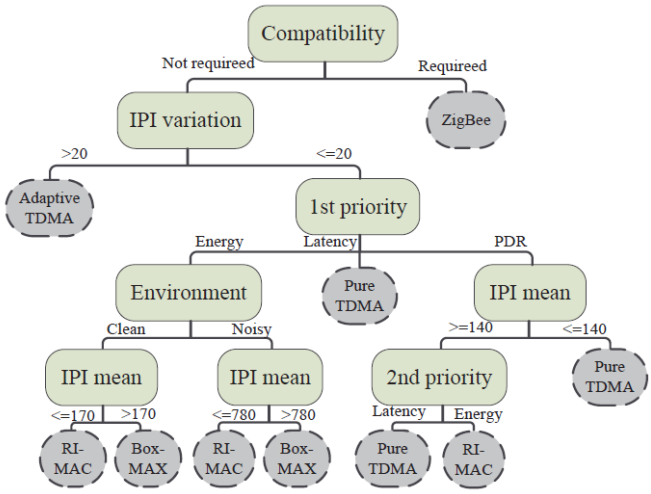


Figure 1.4: A Decision Tree classifier

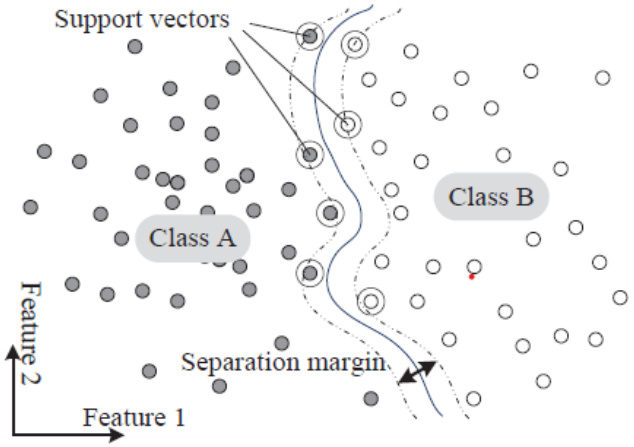


Figure 1.5: Non-linear support vector machines

Figure 1.5 is an illustration of a WSN's data as points in the feature space, with a non-linear SVM as a divider.

Neural Networks and Deep learning:

Neural networks are built by layering multiple chains of perceptrons(neurons), those neurons are decision units that by combining them the algorithm can learn very complex and non-linear functions, deep learning uses NN in-depth for more complex pattern recognition, however this technique is very expensive computationally which limits their usage for WSNs, but with the right model we can optimize the energy consumption in WSNs.

Application: Node localization, Predicting energy consumption of sensor nodes based on various environmental and network-related features. Deep learning models were also

used to optimize routing protocols, data compression, and adaptive sampling techniques in WSNs.

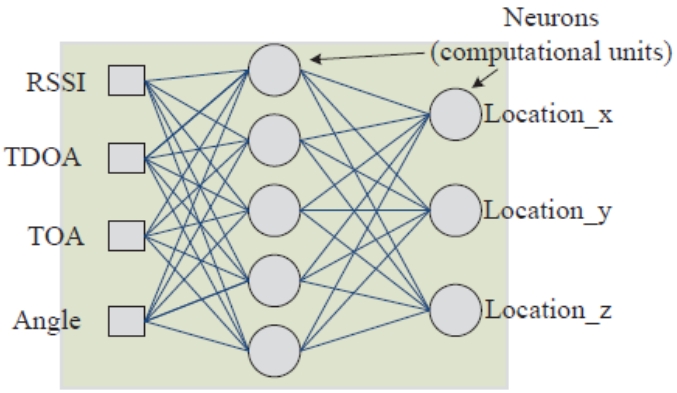


Figure 1.6: A Simple Neural Network For Node Localization

Figure 1.6 shows an example of node localization in WSNs in 3D space using supervised neural networks.

Recent trends in deep learning for WSNs include:

- Convolutional Neural Networks (CNNs) used for spatial feature extraction in sensory data.
- Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks mostly applied for temporal pattern recognition in energy consumption
- Attention mechanisms used to focus on the most relevant features in energy optimization tasks.

1.3.3 Unsupervised Learning Approaches

In unsupervised learning, the model is not given labels. In essence, the goal of supervised learning is to classify the given sample data into groups from the patterns identified in that given sample, this techniques offers an opportunity for energy saving in WSNs without the need for labeled data, the most common approaches are:

K-means Clustering

The algorithm divides the given data points into K clusters, it is widely used in sensor nodes clustering for its simple implementation and linear complexity.

Application: As already mentioned, K-means clustering can be used in grouping sensor nodes with similar energy consumption patterns to implement targeted energy-saving strategies. K-means has also been used for energy-efficient data aggregation and compression in WSNs.

Recent advancements in clustering for WSNs include:

- Adaptive K-means algorithms which automatically determine the optimal number of clusters
- Fuzzy C-means clustering to handle overlapping energy consumption patterns
- Spectral clustering in energy-efficient topology control in WSNs

Principal Component Analysis (PCA)

This is a multivariate algorithm for dimentionality reduction and also data compression by extracting valuable information from the data and then transforming it to a new set of variables known as principal components.

Application: Compressing sensor readings to reduce transmission energy without a great loss in information. PCA has also been used for anomaly detection in energy consumption patterns.

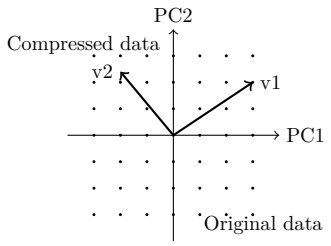


Figure 1.7: PCA for Data Compression in WSNs

Recently PCA was used in WSNs as:

- Kernel PCA for handling non-linear relationships in sensor data
- Incremental PCA for online data compression in streaming sensor data
- Robust PCA for handling outliers and noise in energy consumption data

1.3.4 Semi-supervised learning

Semi-supervised learning uses a combination of labeled and unlabeled data to train the model, this is so useful in WSNs since fully labeled data is hard to obtain in IoT environ-

ments, common semi-supervised techniques include:

Label Propagation

Label propagation works o graph structured data, where node can be labeled and unlabeled, and edges are the relationships between them. The model propagates labels from from labeled nodes to the unlabeled ones based on the similarity between every 2 nodes, we can find this similarity as edge weights, this process is done repetitively until we reach a global consensus.

Application: Classifying energy states of sensor nodes with limited ground truth data. Label propagation has also been used for energy-efficient event detection and localization in limited labeled WSNs.

Co-training

Co-training is a technique where multiple classifiers are trained on different 'views' of the data assuming that each view is independent, each classifier is trained on some of the features, later their predictions of the unlabeled data is used to train other classifiers that complement eachother.

Application: Combining network topology and sensor reading features to predict optimal transmission power levels. Co-training has also been applied to energy-efficient data aggregation and fusion in heterogeneous WSNs.

Graph Neural Networks (GNNs)

GNNs are neural networks created to work specifically on graph-structured data, which makes their implementation in WSNs natural where sensor nodes and their links are easily represented as graphs. GNNs takes into consideration both node features and the graph topology to learn node embeddings.

The graph is represented by and adjacency matrix A that contains the edges and the edges' weights , and the feature matrix X that has the features of every node i.

The hidden representation of a node i at $t+1$ can be obtained from the message passing process which is the sum of the edge weights between the nodes i and every neighbor j multiplied by the transformation function of the representation of the node i at t , all multiplied by a non-linear activation function (like ReLU), this process repeats for T iterations, so that information can propagate across the graph. After T iterations, the node embeddings $H^{(T)}$ can be used for various tasks such as node classification or link

prediction.

Application: In the context of WSNs, GNNs can be used for tasks such as energy-efficient routing, where the graph structure of the network is designed to make intelligent routing decisions. By learning to aggregate information from neighboring nodes, we can predict energy consumption patterns, detect anomalies, and optimize communication in a distributed way.

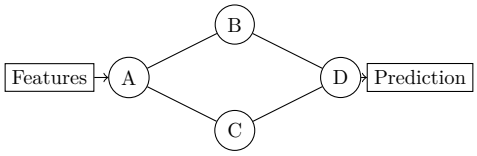


Figure 1.8: A Simple GNN for WSN Optimization

Recent advancements in GNNs, such as attention mechanisms and spatio-temporal GNNs, further optimize their usage in WSNs. Attention-based GNNs assign different weights to different neighbors during the aggregation process, this makes the model focus on the most relevant nodes in the network. Spatio-temporal GNNs, on the other hand, model both spatial and temporal dependencies in the sensory data, which makes the suitable for dynamic WSNs where energy consumption patterns change over time.

1.3.5 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning technique that uses the trial and error process with rewards and penalties to achieve optimal results. Recently, in WSN energy optimization, Rl was widely used due to its capacity to learn the most optimal policies by interacting with the WSN environment, one of the most used RL techniques is Q-learning, where the agent constantly updates the total rewards (Q-value) received from taking an an action a_{t} at a certain state s_{t}, this method can be visualized in Figure 1.9 : **Application:**

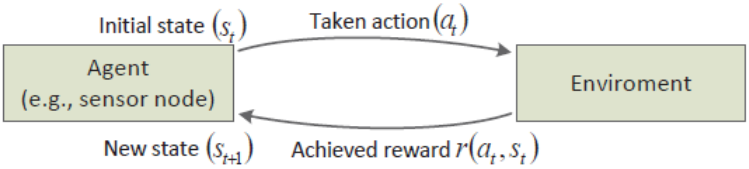


Figure 1.9: Q-learning method

Adaptive duty cycling, where RL agents learn to adjust the sleep/wake schedules of sensor nodes based on network conditions and energy availability. RL has also been used to optimize routing decisions and transmission power control in WSNs.

Recent research in RL for WSNs brought advancements including:

- Deep Reinforcement Learning (DRL) combining deep neural networks with RL for handling high-dimensional state spaces
- Multi-agent Reinforcement Learning (MARL) for coordinated decision-making among multiple sensor nodes
- Model-based RL approaches that learn a model of the environment to improve sample efficiency

1.3.6 Challenges of ML in WSNs:

While the necessity of deploying ML models in WSNs is evident for its significant improvement in minimizing the energy consumption, it is necessary to acknowledge the challenges and difficulties so we can address them in future works. One of the main problems is the resource constrained environments, due to the resource limited nature of sensor nodes, it is hard to implement complex ML models in such environments, in the other hand, the ML models face a challenge in adapting to dynamic network conditions and energy patters, future works should also address the scalability problem to ensure that the ML solutions can adapt to large scale WSN deployments, and while ML models deliver outstanding results, they are often seen as black boxes, so it is important to make those solutions understandable to allow further advancements from other researchers.

Conclusion

In this chapter, we provided all the necessary basic knowledge in the scope of WSNs and energy efficiency and we also reviewed various strategies to improve it, from traditional techniques like duty cycling and data aggregation to advanced ML-based methods. While these approaches are effective in reducing energy consumption, their success often depends on challenges like limited computational resources and network dynamics. Combining traditional approaches with ML-based techniques could offer significant solutions for improving the sustainability and reliability of WSNs in the years ahead.

Chapter 2

Literature Review

This chapter aims to provide a comprehensive overview of the most recent approaches used for energy conservation in WSNs and IoT application with emphasis on duty cycling optimization. This analysis will serve as a foundation for understanding ML and DL architectures that were proposed by various researches, in which we will review each method and propose a critical review of it in this thesis.

2.1 Energy Efficiency in Reinforcement Learning for Wireless Sensor Networks:

In this paper, M.Kozlowski and R.Mcconville proposed an innovative approach tackling energy-effecient indoor localization using RL techniques in WSNs. The method relies in its core on its formulation as a MDP and the application of SARSA algorithm for continuous weak training, in the following sections we will explain the innovation brought by this paper and the key models and techniques used in this work.

2.1.1 Markov Decision Process (MDP) Formulation:

The problem is formulated as an MDP with two states: S1: Enhanced sensing state, and S2: Low-power sensing state. With two actions possible in each state:

A1: Stay in the current state, and A2: Switch to the other state

They also used a simple reward function that is designed to encourage energy efficiency while also maintaining performance:

$$r(s_t, a_t) = \begin{cases} -1 & \text{if } s_t = S1 \text{ and } a_t = A1 \\ +1 & \text{if } s_t = S2 \text{ and } a_t = A2 \\ 0 & \text{otherwise} \end{cases} \tag{2.1.1}$$

This reward function penalizes staying in the energy-intensive state (S1) and rewards staying in the low-power state (S2). And to ensure the localization performance they added an additional performance boost B_t :

$$B_t = \begin{cases} -1 & \text{if } e(t) \geq e(t-1) \\ +1 & \text{if } e(t) < e(t-1) \end{cases} \tag{2.1.2}$$

where e(t) is the localization error at time t. This boost aims to encourage the system to improve or maintain localization accuracy even when in the low-power state.

2.1.2 SARSA Algorithm:

The authors also employed an on-policy temporal difference learning method by using the SARSA algorithm, the objective is to learn the optimal policy, with the Q-value update rule :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[B_t + r(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.1.3}$$

where: α is the learning rate, γ is the discount factor, $Q(s_t, a_t)$ is the Q-value for state s_t and action a_t, B_t is the performance boost and $r(s_t, a_t)$ represents the immediate reward

The performance boost B_t included in the Q-value update allows the system to balance energy efficiency and localization accuracy dynamically.

2.1.3 Action Selection Methods:

Three action selection methods are used in this paper to find the trade-off between exploitation and exploration:

1. Greedy: It always selects the action with the highest Q-value.

$$a_t = \arg\max_a Q(s_t, a) \tag{2.1.4}$$

2. -Greedy: Selects the greedy action with probability 1- , and a random action with probability .

$$a_t = \begin{cases} \arg\max_a Q(s_t, a) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases} \tag{2.1.5}$$

3. **Softmax:** Selects actions probabilistically based on their Q-values.

$$P(a_t = a) = \frac{e^{Q(s_t,a)/\tau}}{\sum_b e^{Q(s_t,b)/\tau}} \tag{2.1.6}$$

where is the temperature parameter controlling the exploration-exploitation trade-off.

2.1.4 Hidden Markov Model (HMM) for Localization:

For localization inference, M.Kozlowski and R.Mcconville employ a Hidden Markov Model (HMM) approach. The joint probability of locations x_t and observations z_t is given by:

$$p(x_{1:t}, z_{1:t}) = p(x_0) \prod_{i=1}^{t} p(z_t|x_t)p(x_t|x_{t-1}) \tag{2.1.7}$$

where:

- $p(x_0)$ is the initial state distribution
- $p(z_t|x_t)$ is the emission probability
- $p(x_t|x_{t-1})$ is the transition probability

The emission probabilities are modeled as Gaussian distributions:

$$p(z_t|x_t) = \sum_{k=1}^{K} \mathcal{N}(z_t|\mu_{jk}, \sigma_{jk}) \tag{2.1.8}$$

where j are the location states and k is the access point sensors. This Gaussian model allows a dynamic representation of the signal strength distributions at every location.

Algorithm Implementation:

The authors proposed an algorithm that combines the MDP formulation, HMM based localization and SARSA learning to ensure an energy efficient adaptive localization, the algorithm can be outlined as follows:

2.1.5 Simulation and Performance:

The method was validated on the SPHERE challenge dataset that has 4 unique access points, location labels and sensors that can be used as ”oracles”, the model was trained on 15

Algorithm 1 HMM and SARSA Algorithm

```
Initialize HMM and SARSA parameters
for each time step do
    Collect sensor observations
    Perform localization inference using the HMM
    if in enhanced sensing state (S1) then
        Collect weak labels from oracle sensors
        Re-estimate HMM parameters
        Calculate localization error
    end if
    if in low-power state (S2) then
        Retain previous error estimate
    end if
    Calculate performance boost
    Select next action using chosen selection method (Greedy, ϵ-Greedy, or Softmax)
    Update Q-values using SARSA
    Transition to next state based on selected action
end for
```

The simulation state space size varied between 10 to 30 states with a variable number of access points, and the BLE path loss model was used to simulate signal strength, after deploying the solution in this environment, the experiment showed a good performance on the validation results, with the greed method improved from 4.8m error (Control) to 2.5m error (Reinforced) and it has also reduced the dependence on energy-inefficient sensors to 0, while the -Greedy method showed improvement from 4.8m error (Control) to 2.1m error (Reinforced) and a 16% dependence after 100% of iterations , finally, the Softmax method Improved from 4.8m error (Control) to 1.8m error (Reinforced) with a 35% dependence after 100% of iterations.

This shows that the algorithm successfully reduced dependence on energy-inefficient sensors while improving localization accuracy, with the -Greedy method showing the best results for the action selection, which offers a good trade-off between sensor usage and performance improvement.

2.1.6 Critical review:

While the paper presents an innovative approach by formulating the problem as an MDP with energy aware states and combining RL techniques with traditional localization methods, also the idea of the continuous weak training is impressive an SARSA was well suitable

for it.It is important to note the limitations of this method, to be able to further enhance it.

First, the solution is not well suitable for real life application in energy efficient indoor localization, we can deduce this from the simulation and validation environment of the RL agent, where we have the dependence on the oracle sensors, which are high-accuracy but energy-intensive sensors, which is not always the case, also the method does not account for dynamic changes in the environment (e.g furniture moves, new obstacles..) which may affect the RSS patterns, the paper also states that the real world validation only achieves room-level accuracy which is not enough for many applications, Furthermore, the simulation only takes up to 30 states which is relatively small for actual environments.

Aside from the real world application, the method also shows some limitations in terms of methodology, the method was primarily evaluated on simulated data, with only limited validation on a single real-world dataset (SPHERE Challenge) , the performance appears to be sensitive to many parameters (learning rate, discount factor, oracle weights, ...) which were chosen empirically .Also, the method was not compared against other state of art adaptive or energy aware localization techniques, which makes it difficult to evaluate its performance compared to other methods performing the same task, the method could also explore using loss models other than a basic BLE path loss, to fully capture the details and complexities of real indoor environments.

Despite these limitations, this work provides a solid theoretical foundation for future research in energy-aware adaptive localization systems.

2.2 GSAVES:

In this paper, Laidi et al. presents GSAVES (Graph Sensor AdVersarial for Energy Saving), a new approach for maintaining sensor energy in event-based Internet of Things (IoT) applications. The proposed method aims to preserve energy in both sensing and communications by deactivating a portion of IoT devices while using readings from active sensors and the network's spatial correlation to generate missing data. In this solution no duty cycling scheduling is required since the sensor deactivation process is random, meanwhile an adversarially trained graph Convolutional network (GCN) generates the missing data of the sleeping sensors.

The core idea is to extend the network's lifetime without impacting the network's reliability, and that is by accurately estimating the readings of the sleeping sensor after learning the spatio-temporal characteristics of the sensor network . The GSAVES model operates at the data collector level (cloud, base station, or edge), where the generator is proposed to generate the missing data of sleeping sensors. Figure 2.1 illustrates the

Overview of the proposed solution in this paper.

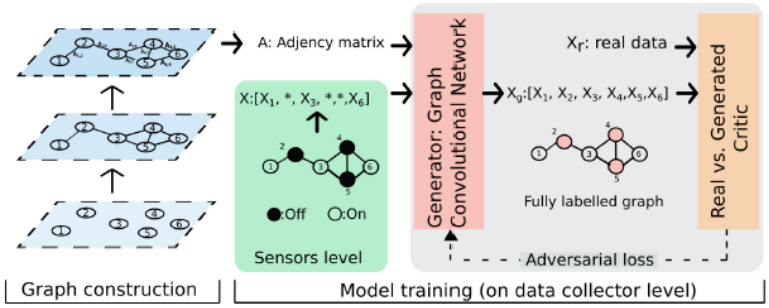


Figure 2.1: General Architecture of GSAVES

The network is modeled as a graph, to capture the spatial relationships between the sensor nodes, and the generator is a graph convolutional network trained adversarially in a semi-supervised manner against a critic network, where the generator replaces the missing data from the sleeping nodes while the critic learns to distinguish between the imputed and the real data. A concrete formulation of the problem would be: Let $X \in 0, 1, *^{N \times d}$ denote the matrix representing sensor data, with N being the sensor network size, and d the number of node attributes representing the length of historical readings. The values 1 and 0 indicate the occurrence or absence of an event, while the symbol * denotes a missing value caused by inactive sensors. The generator receives the adjacency matrix A, representing the mutual coverage between sensors, calculated in the pre-processing phase. A binary mask matrix $M \in 0, 1^{N \times d}$ is defined to express the missing data in X, where each row m_i indicates the presence or absence of reading in x_i. The generator exploits the information in X, M, and the spatial relationships among the sensors described by A to fill in the missing values in X and create a complete matrix $\tilde{Y} \in 0, 1^{N \times d}$ of the generated data. Formally, the generator learns a mapping function f, defined as:

$$\tilde{Y} = f(X, M, A) \tag{2.2.1}$$

The final output $Y \in 0, 1^{N \times d}$ of inputted missing values is then calculated using:

$$Y = M \odot X + \bar{M} \odot \tilde{Y} \tag{2.2.2}$$

where $\bar{M}$ is the logical binary complement of M and $\odot$ denotes the Hadamard product. The generator is implemented as a Graph Convolutional Network (GCN) that can extract knowledge from graph structures. The critic's role is to challenge the generator during training to produce synthetic data samples closer to the real distribution. Using the W-GAN approach, the goal of the critic is to discover a function c that allows the estimation of the Wasserstein (or Earth-mover) distance between the distribution of the actual data p_r and the distribution of the generated data p_g. The Wasserstein distance is given by:

$$L_c = \sup_c \mathbb{E}x \sim p_r[c(x)] - \mathbb{E}\tilde{x} \sim p_g[c(\tilde{x})] \tag{2.2.3}$$

For performance evaluation, intelligent buildings were chosen as a use-case scenario. They used the MERL dataset, a real dataset collected indoors that includes over 50 million motion sensor events spanning over two years with milliseconds granularity. The experiments compared the proposed solution with four state-of-the-art approaches: GCN, GAIN, GINN, and JGD. They have also evaluated the impact of the sleeping nodes' percentage on data accuracy, varying this percentage from 10The results showed that GSAVES provides the highest accuracy compared to the other solutions, with its F-score slightly improving with the number of nodes. This improvement is attributed to the model's capacity to learn from a larger context, as a higher number of sensors allows the model to better approach global data distribution. For energy performance evaluation, this work considered the average energy consumption per node in terms of milliwatt-second (mWs) while varying the accuracy rates. The energy consumption for a sensor i throughout d time slots was calculated as:

$$E_i = \sum_{j=1}^{d} y_{ij} m_{ij} \; e_{stb}(\sum_{j=1}^{d} m_{ij} - \sum_{j=1}^{d} y_{ij} m_{ij}) \; e_{inact}(d - \sum_{j=1}^{d} m_{ij})] \tag{2.2.4}$$

The experimental results consistently showed that GSAVES guarantees the most extended lifetime among the compared solutions while maintaining the highest accuracy.

2.2.1 Critical Review:

After Carefully examining this work, it was easy to see the innovation in this work compared to other works with the same aim, GSAVES offers a promising solution for the challenge of energy conservation, by combining random sleep scheduling with the data generation, we can clearly see how both the sensing and communication energy consumption are addressed.

However, there are some limitations and areas for improvement: This paper tested the solution on a network of 32 sensors, in IoT networks we have networks with a much higher number of nodes, and it is not clear how GSAVES would perform in such networks. Similarly, the model was only trained on one graph topology assuming a static sensor network, it would have been interesting to test GSAVES on dynamic environments where the relationships between the sensors change overtime. Another important aspect is the security consideration, since the data is generated synthetically, the system is exposed to attacks that manipulate the generator to produce false readings. As for the energy consumption evaluation, a more detailed energy model that considers factors like transmission power, variable sensing rates, or environmental conditions could provide more accurate estimates of energy savings.

Nevertheless, GSAVES is a significant step in energy-efficient IoT sensor networks, the combination of graph neural network with adversarial training is an innovative approach

to address energy problems in sensor networks, this work represents a foundation for other future solutions bringing more adjustments and enhancements for a better generalization.

2.3 On Predicting Sensor Readings With Sequence Modeling and Reinforcement Learning for Energy-Efficient IoT Applications:

In this paper, we are introduced to a model that is capable of learning long and short-term spatio-temporal relationships in sensory data and predict future values. This model enables turning sensors off for extended periods to preserve energy. The proposed architecture consists of three main components: LSTM network, RL agent and physical sensors. The LSTM model and physical sensors work together to monitor the network. When the sensors are on, the system relies on their readings. However, the LSTM outputs predictions of the readings when sensors are turned off, On the other hand the RL agent acts as a supervisor here, deciding when to use physical readings or LSTM estimation, balancing power preservation and detection accuracy. Figure 2.2 shows the general framework of the proposed solution in this paper.

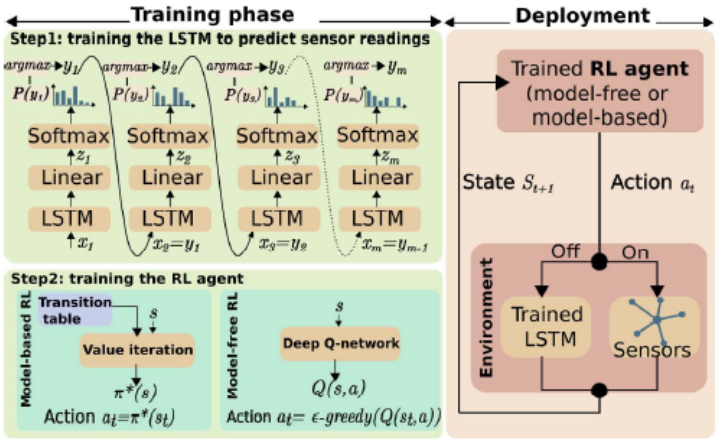


Figure 2.2: General Framework

2.3.1 LSTM Model for Predicting Sensor Readings:

In this work, an LSTM network is used to learn complex spatiotemporal patterns found in sequential sensor data. The goal is to provide an accurate future event prediction by learning from the past sensor readings using the LSTM's capability of modeling both short-range and long-range dependencies.

This is done first by considering sensor data in pairs of input and output sequences over time (X, Y) . The inputs are the sensor readings and their forecasted values along a timeline. In other words, the LSTM learns iteratively from its previous predictions and surroundings to predict the probability of future states, leveraging an extended knowledge of past events for more accurate predictions.

The LSTM's model architecture that lets the network to keep information for some time in its hidden state presents a great advantage. This allows the network to analyse longer sequences of readings, this is particularly useful for tasks involving predicting future sensor states based on historical data. These predictions are made using the softmax layer, which outputs a probability distribution over possible outcomes so that the network can select the most likely future state.

2.3.2 Reinforcement Learning Agent for Optimal Prediction Sequence Length:

In addition to the LSTM, a reinforcement learning (RL) agent is proposed to further optimize the prediction sequence length by determining which sensors should be in the active state. The RL algorithm uses a MDP, where the state is the current readings of the sensors, and the two actions are turning sensors on or off. The RL agent's objective is to maximize prediction accuracy while minimizing energy consumption by selectively controlling the sensors.

The authors propose two approaches to train the RL agent: model-based (LSTM-DP) and model-free (LSTM-DQN). The model-based method uses the value iteration algorithm to compute the optimal policy by evaluating all possible state transitions and actions. But, because of the high computational costs in this method, the paper also propose a model-free solution using Deep Q-Networks (DQN). The DQN approximates the action-value function through a NN, enabling better scalability to larger WSNs.

By combining LSTM for sequence prediction and reinforcement learning for sensor optimization, the authors effectively address both prediction accuracy and energy efficiency in sensor networks, offering a comprehensive solution for managing large-scale, energy-constrained systems.

2.3.3 Experimental Setup and Results:

Here again, the MERL dataset is used for the model training and evaluation, The experiments focused on three aspects: Prediction accuracy, Energy consumption and scalability. Both model, LSTM-DP and LSTM-DQN were compared to two state of the art solutions: Joint Gaussian Distribution (JGD) and Event Correlations (EC). In addition to that,

the authors investigated the impact of learning spatio-temporal correlations by comparing against DNN variants (DNN-DP and DNN-DQN) and a baseline solution (LSTM-IP). The results showed that both LSTM-DP and LSTM-DQN outperformed the state-of-the-art solutions (JGD and EC). The proposed solutions achieved about 50

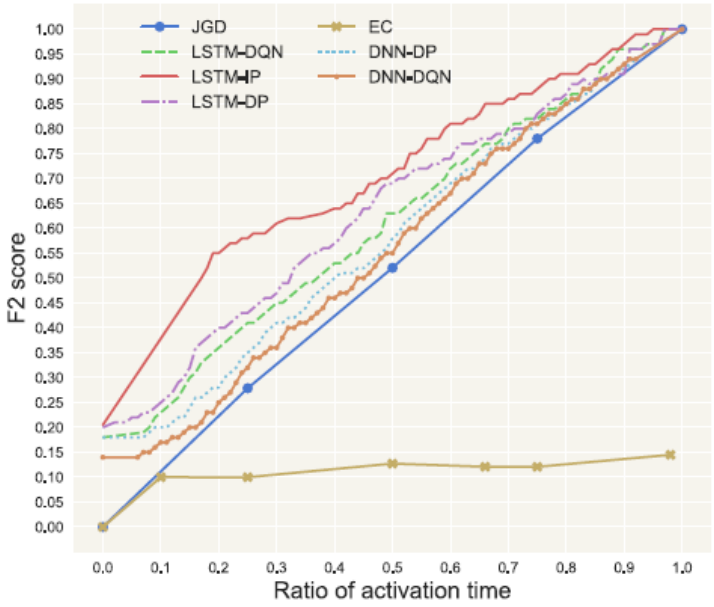


Figure 2.3: F2-score on the test set for four sensors as a function of the ratio of energy.

As for the energy consumption, the experiment consisted of measuring the energy consumed by one sensor while varying the accuracy rate, the final results showed that LSTM-DP achieved better energy savings than LSTM-DQN, and both outperformed JGD.

The scalability of the solution was investigated by varying the number of sensors for 4 to 29, the performance was mesured by: F-score, Latency, Network's lifetime, Average energy consumed by an active sensor, and the ratio of missed events. The experiment shows that extending the network's size increased the F2-score and reduced the ratio of missed events. It also reduced energy consumption and expanded the lifetime. However, increasing the number of sensors raised the latency.

2.3.4 Critical Review:

This work presents a promising approach for energy-efficient sensor reading prediction in IoT applications. The integration of LSTM for spatiotemporal pattern learning and RL for decision-making is a flexible and effective solution. However, this work also has some limitations that should be addressed to further enhance the results and performance of this solution, like the assumption of a stationary environment, while in a real-case scenario we

are often faced with non-stationary environments in IoT networks. The increased latency with the number of sensors is also an issue to be addressed by possibly using parallel processing for time-sensitive applications.

On another point, the scalability of the LSTM-DP model is limited to only 8 sensors, which is probably not useful in real world applications, and for a further generalization of the solution, it would be interesting to evaluate the models on datasets other that the MERL dataset, for different domains that would also allow an extension to a multi-class event detection or continuous value prediction.

Finally, it is important to acknowledge the evaluation that demonstrates significant improvements over existing methods in terms of prediction accuracy and energy consumption.

2.4 PriSTI: A Conditional Diffusion Framework for Spatiotemporal Imputation:

Liu et al. presented in this paper the PriSTI framework, the problem addressed by this framework is relevant in various domains like air quality monitoring, traffic forecasting and other problems that require a spaito-temporal data imputation. The missing data is caused by the sensor failures or data transmission losses, after examining other methods the authors came out with this framework that should address the shortcomings and enhance the data imputation in datasets that have dependencies across both space and time. While the paper does not address directly energy conservation in WSNs, the imputation process proposed here can be accustomed for energy-saving techniques in IoT applications.

2.4.1 Methodology:

PriSTI relies on diffusion probabilistic models (DPMs) for spatiotemporal data imputation, the task at hand is presented as a conditional generation problem. DPMs have shown outstanding success in image composition and audio generation, the authors chose to use them here in a two-stage process: the diffusion process and the reverse process.

Diffusion Process: In this stage, a Gaussian noise is progressively added to the observed spatiotemporal data, making it noisier with time. The goal is to corrupt the data to a point where it become similar to a standard normal distribution. This process can be described as follows:

$$q(X_{1:T}|X_0) = \prod_{t=1}^{T} q(X_t|X_{t-1}), \tag{2.4.1}$$

where $q(X_t|X_{t-1}) = \mathcal{N}(X_t; \sqrt{1-\beta_t}X_{t-1}, \beta_t I)$, with β_t being a small hyperparameter that controls the variance of the added noise. As T increases, the noisy data approaches a

standard normal distribution.

Reverse Process: Once the data is noisy enough, the reverse process aims to recover the clean data by iteratively removing the noise. A noise prediction network, informed by the observed data and the geographic relationships among the sensors, guides this process. The reverse process is described as:

$$p_\theta(X_{0:T-1}|X_T, X, A) = \prod_{t=1}^{T} p_\theta(X_{t-1}|X_t, X, A), \tag{2.4.2}$$

where X represents the observed spatiotemporal data and A is the adjacency matrix capturing the geographical information. A conditional feature extraction module is introduced in the paper to capture complex spatiotemporal dependencies. This module processes the noisy data representations, using interpolated data as a global context prior. Then the model predicts noise at each step, weighted by spatial and temporal attention taken from both the data and the geographical informations. The main contributions brought by this paper are, the Conditional diffusion framework that allows the incorporation of the spatial dependencies and the observed values. In the reverse diffusion process a key innovation is observed with the Noise prediction model, It is a crucial part that predicts and removes noise from the corrupted data by using observed data and geographic adjacency, progressively improving the imputed values. Furthermore, PriSTI provides a robust reference point for capturing spatiotemporal patterns by the idea of using interpolated data as a global context.

2.4.2 Training and evaluation:

The training process proposed consists of masking portions of the observed data and training the noise prediction model to learn how to remove the noise, in more detail; we start by masking the data to create the target imputation data. This data is then added to generate conditional information that will guide the model. Finally, Noise is added to the data in several steps, and the model is trained to predict and remove the noise using a loss function that measures the difference between the actual noise and the predicted noise.

After training, the model imputes missing values into the noisy imputation target sampled from a normal distribution, Then, The model gradually removes the noise in a reverse process, recovering the missing data by sampling values from predicted distributions at each time step.

The paper outlined 3 real world datasets that were used for the model evaluation: AQI-36 (air quality data), METR-LA, and PEMS-BAY (traffic data). The results show that PriSTI outperforms several baseline methods, including traditional statistical models, machine learning approaches, and other deep learning-based imputation models, across

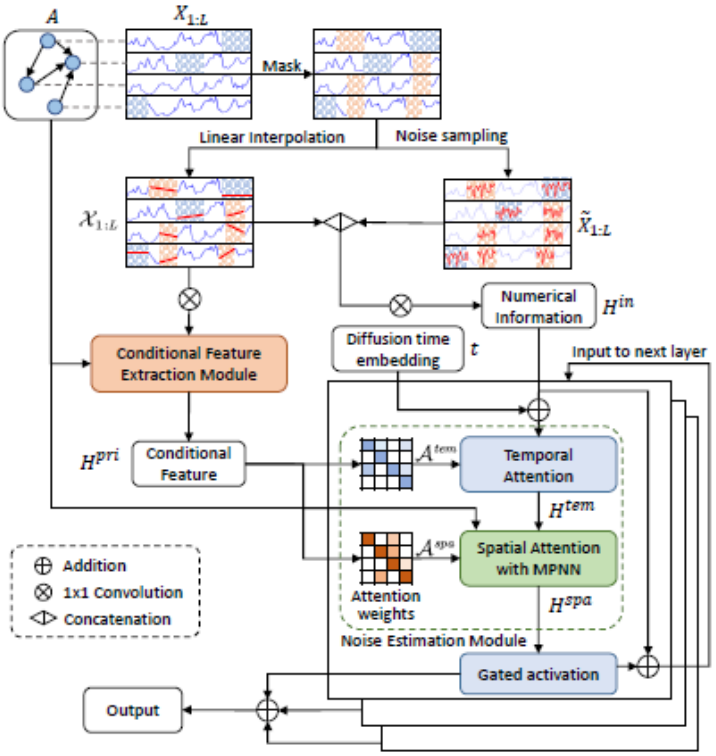


Figure 2.4: The pipeline of PriSTI.

various missing data patterns, It is easily shown that PriSTI excels in scenarios with high missing rates and sensor failures, where other models struggle. The evaluation was carried out using metrics such as MAE and CRPS. For example, on the AQI-36 dataset, PriSTI achieved a MAE of 9.03 for simulated failure patterns, which outperforms the CSDI model, which had a MAE of 10.56. On the other hand, on the PEMS-BAY dataset, PriSTI achieved a CRPS of 0.0064, while CSDI scored 0.0067.

2.4.3 Critical Review:

This paper brought an innovative approach by using generative AI for spatiotempral data imputation, it also introduced various key algorithmic additions, addressing the spatial relationships in sensor networks, an aspect which is often neglected in other studies.

However, sine PriSTI depends on Gaussian noise processes and interpolation-based conditional information some limitations are introduced. First, while interpolation is efficient, it does not always capture the complex non-linear patterns present in real-world data, especially for datasets with highly dynamic spatiotemporal relationships. Additionally, the computational complexity of PriSTI, particularly in terms of training and inference time, is higher than some existing methods, making it less practical for large-scale real-time

applications, and impractical for sensor networks due to their low computational capacity.

Finally, while PriSTI have good results in scenarios with high missing rates, its performance on datasets with low missing rates does not really outperform other methods, so we can say that the strengths of this framework are most useful in particularly challenging imputation tasks.

2.5 TARNet: Task-Aware Reconstruction for Time-Series Transformer:

In this paper, Ranak Roy Chowdhury et al. introduce TARNet, an approach to improve time series classification and regression tasks that relies on task-aware reconstruction. This innovative approach is used to address issues faced by the other state of art methods in deep learning that struggle with semi labeled data.

The authors found that unsupervised pre-training through reconstruction was explored as a way to leverage unlabeled data, but methods like Time Series Transformer (TST) use random masking, which may ignore that certain timestamps are more important for a certain task. For which, they introduced task-aware reconstruction, where the masking strategy is guided by information from the end task.

While this paper does not address the energy conservation task specifically, it is a novel approach for time series classification which can be explored for energy saving in WSNs.

2.5.1 Methodology:

Ranak Roy Chowdhury et al. start with a multivariate time series $X \in \mathbb{R}^{S \times N}$ with S timesteps and N variables, along with a target label y, the goal of TARnet is to predict labels $\tilde{y}$ for unseen data X.

The core of the model is a transformer encoder used starting with a mean standardization of input features, then the feature vectors are linearly projected, followed by an addition of positional encoding then, the processing is done in transformer encoder blocks, finally the weighted outputs are passe through a feed-forward network.

The architecture of TARnet consists of three components: 1. End Task (T_{END}): A standard supervised task used for classification and regression, the model uses the vector from the last timestamp with two fully connected layers with ReLU activation and finally, An output layer.

For classification, the authors applied a softmax to obtain class probabilities, with a Cross-entropy loss for classification and a Squared error loss for regression.

Task-aware Reconstruction: This is an unsupervised reconstruction task with task-informed masking, it aims to recover the input data X after masking. the most important used notions are:

1. A binary mask $m \in \mathbb{R}^S$ generated by the masking strategy M.
2. Masking involves replacing feature vectors with zeros at selected timestamps.
3. The masked input is processed through Transformer Encoder layers.
4. The output is passed through fully-connected layer to get the reconstructed data $\tilde{X}$.

The reconstruction loss L_{TAR} is a weighted sum of masked and unmasked losses, and the total loss is a combination of the end task and reconstruction losses.

Data-driven Masking Strategy (M): This mechanism is used to select important timestamps for masking based on final task characteristics, this is the one of the innovation brought by this paper, this strategy uses self-attention weights from the end task to identify important timestamps, the proposed process goes like:

1. Computing aggregate attention map $A \in \mathbb{R}^{S \times S}$ from Transformer Encoder layers.
2. Calculating normalized aggregate attention weights $\sigma \in \mathbb{R}^S$: $\sigma_k = \frac{\sum_{i=1}^{S} A_{ik}}{\sum_{k=1}^{S} \sum_{i=1}^{S} A_{ik}}$
3. Selecting top $\lfloor \beta S \rfloor$ values from σ to form σ'.
4. Randomly sampling $\lfloor \mu S \rfloor$ timestamps without replacement from σ' to generate m.

This approach ensures that important timestamps for the end task are masked and reconstructed with Randomization added to prevent overfitting to a fixed set of timestamps.

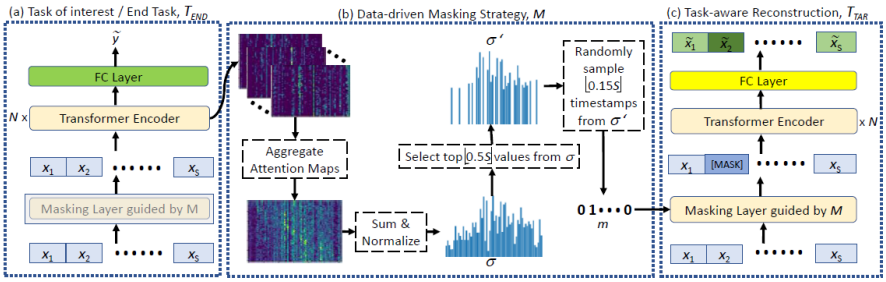


Figure 2.5: Overview of TARnet.

2.5.2 Training and evaluation:

The authors evaluate TARNet on a diverse set of time series classification and regression tasks, comparing it against numerous statistical and deep learning baselines, unsing numerous data benchmarks covering various domains (e.g., motion, audio, EEG, HAR) from UEA Archive, UCI Machine Learning Repository, Monash University, UEA, UCR Time

Series Regression Archive.

The complete training procedure for TARNet is shown in Algorithm 1

Algorithm 1 Training of TARNet

Input: X, y
Hyper-parameters: μ, β, λ, η
Output: *Model*

σ initialized randomly
Model = TransformerEncoder()
while training **do**
  σ' = top $\lfloor \beta S \rfloor$ values from σ
  $m \sim$ Randomly sample $\lfloor \mu S \rfloor$ timestamps without replacement from σ'
  $\tilde{X}, \tilde{y}$, A = *Model*.train(X, m) # A ← Self-Attention Scores
  Compute $\mathcal{L}_{TAR}(\tilde{X}, X, \lambda)$ and $\mathcal{L}_{END}(\tilde{y}, y)$
  $\mathcal{L}_{Total} = \eta \mathcal{L}_{TAR} + (1 - \eta)\mathcal{L}_{END}$
  σ = add_and_normalize(A)
end while
return *Model*

Figure 2.6: TARnet training.

For the sake of evaluation, the authors compared TARnet to numerous methods and baselines, including Time Series Transformer (TST), TS2Vec, TNC, TS-TCC, ResNet, ShapeNet and WEASEL-MUSE, Using various metrics for a fair evaluation including: 1. Accuracy (for classification) and RMSE (for regression) 2. Ours 1-to-1 Wins/Draws/Losses: Number of datasets where TARNet performs better/same/worse than baselines. 3. Mean Rank: Average rank of each model across datasets. 4. Avg.Rel.Diff.Mean: Average relative difference from mean performance across datasets

The results of the comparison TARnet show superiority to other methods, with the highest average accuracy (77.2 Similarly TARnet showed great results for regression tasks,

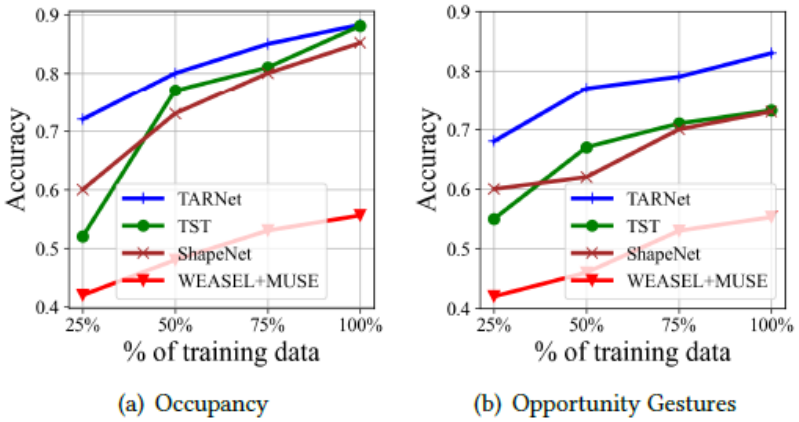


Figure 2.7: TARnet classification results.

ranking first on three datasets and second on two datasets, outperforming all baseline

models, with the lowest mean rank (1.833) across all datasets, compared to 2.5 for the next best model (TST), and a 31.3

2.5.3 Critical Review:

After reviewing this paper, it is safe to say that TARnet presents a novel and promising approach to improving time series analysis tasks, the authors addressed previous problems in models performing the same task as TARnet and they delivered a comprehensive evaluation showing how their model outperforms other methods, that is die to their Innovative masking strategy and the flexibility of the architecture and an improved data efficiency.

However, this work shows some limitation due to the potentially very high computational complexity of the model compared to the other baseline models, which makes this approach highly inappropriate for energy conservation in IoT environments and especially for WSNs, also, this method introduced many other new hyperparameters which may make the highly model sensitive leading to lower performance in simpler environments like WSNs, additionally , a further generalization of this method is needed in other architectures to see how task-aware reconstruction performs.

In conclusion, TARNet represents a significant contribution to the field of time series analysis, introducing a novel approach that combines supervised and unsupervised learning in a task-aware manner, however due to the limitations that we already sited, this method is not suitable for energy aware conservation tasks in WSNs.

Part II

Contribution

General Introduction

2.6 Background and Motivation

Wireless Sensor Networks (WSNs) are vital for applications like environmental monitoring, disaster management, and industrial automation, thanks to their ability to gather, process, and transmit data across distributed sensors in real-time. However, the energy constraints of battery-operated sensor nodes, especially in remote and dynamic environments, present a significant challenge in prolonging network lifetime while ensuring data quality. A major hurdle in current WSN approaches is the fragmented treatment of spatial and temporal data dependencies. While graph-based methods model the spatial relationships between sensors and time-series techniques handle temporal patterns, these solutions often assume uniform sensor reporting intervals and overlook the irregularities common in real-world deployments, such as sensor failures or energy-saving outages. Furthermore, centralized data processing raises privacy and communication concerns, especially in large-scale networks. Federated Learning (FL) offers a promising decentralized alternative, addressing these issues by enabling local model training across sensor nodes, reducing communication overhead and enhancing data privacy.

2.7 Problem Statement

The problem addressed in this thesis revolves around the efficient modeling of spatiotemporal dependencies in WSNs while accounting for irregular time intervals and the computational constraints of the network. Current solutions either treat the problem as a graph optimization task or a time series modeling challenge, often overlooking the irregular temporal nature of sensor data. Moreover, many high-performing models require centralized data processing and significant computational power, both of which are impractical for real-world WSN applications where data privacy, security, and energy efficiency are paramount.

2.8 Research Gap

Our research builds on a comprehensive review of existing methods in WSNs, where we identified key shortcomings in addressing the irregular time intervals between sensor readings. Most existing models fail to integrate both spatial and temporal dependencies effectively, and those that do often overlook the varying temporal gaps in real-world deployments. Additionally, current centralized training methods introduce significant communication overhead and security risks, which further constrain the energy efficiency and scalability of the system. These gaps in existing research form the foundation for our work.

2.9 Objectives of the study

The main objective of this thesis is to propose and evaluate an innovative solution that integrates both spatial and temporal aspects of WSN data while specifically addressing the irregular time intervals between sensor readings. Our proposed Delta-GCN model combines Graph Convolutional Networks (GCNs) with a customized Long Short-Term Memory (LSTM) layer to capture spatial and temporal dependencies. This novel model is designed to operate efficiently in resource-constrained WSNs by minimizing computational complexity and energy consumption. Furthermore, we incorporate a decentralized Federated Learning (FL) framework, ensuring that data remains distributed across the network, reducing communication overhead, enhancing data privacy, and optimizing energy usage.

2.10 Structure of the report

This thesis is structured as follows: In the first chapter, we review the state-of-the-art methods for energy-efficient WSNs, focusing on graph-based and time-series approaches. The second chapter introduces our proposed Delta-GCN model, detailing its architecture, training process, and integration with the FL framework. Chapter three presents the empirical evaluation of our model, evaluating its performance using key metrics such as energy consumption, precision, recall, and network lifetime. Finally, we conclude with a discussion on the implications of our findings and potential future research directions.

Chapter 3

Research and Conception

3.1 Theoretical Foundation

3.1.1 Graph Convolutional Networks (GCN)

Graph Convolutional Networks (GCNs) are powerful tool for learning on graph-structured data, extending the concept of convolutional neural networks to non-Euclidean domains. At their core, GCNs are designed to capture and empower the relationships between nodes in a graph, for this particular reason and because we chose to represent our data in the form of graphs, we chose to implement them at the core of our model structure, to capture the spatial relationships in the WSN.

Neural Networks

GCNs are built upon the fundamental principles of neural networks. Like traditional neural networks, they consist of layers of interconnected nodes that transform input data through a series of non-linear operations. However, GCNs introduce a crucial modification: the concept of graph convolution.

In a standard neural network, each node in a layer is connected to all nodes in the subsequent layer. In contrast, GCNs restrict these connections based on the graph structure, allowing information to flow only between adjacent nodes in the graph. This locality principle is key to capturing the spatial relationships within the graph structure

Convolution Operation

The core operation in GCNs is the graph convolution, which can be thought of as a generalization of the convolution operation used in image processing to graph-structured data.

In its simplest form, the graph convolution operation for a node can be expressed as:

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \frac{1}{c_{ij}} h_j^{(l)} W^{(l)}\right)$$

Where:

- $h_i^{(l+1)}$ is the feature vector of node i at layer $l+1$
- $\mathcal{N}(i)$ is the set of neighbors of node i
- c_{ij} is a normalization constant (often the square root of the product of the degrees of nodes i and j)
- $h_j^{(l)}$ is the feature vector of node j at layer l
- $W^{(l)}$ is the weight matrix for layer l
- σ is a non-linear activation function

This operation aggregates information from a node's neighbors, applies a learned transformation, and then passes the result through a non-linearity. The process can be viewed as a form of message passing between nodes in the graph

Forward Pass in GCNs

During a forward pass in a GCN, the following steps occur:

1. Feature Propagation: For each node, the features of its neighboring nodes are gathered.

2. Aggregation: These neighbor features are combined, often through a simple sum or mean operation.

3. Transformation: The aggregated features are multiplied by a learned weight matrix.

4. Activation: A non-linear activation function is applied to introduce non-linearity into the model.

5. Update: The node's features are updated with this new representation.

These steps are repeated for each layer in the GCN, allowing the network to capture increasingly complex patterns in the graph structure.

Adjacency Matrix and Edge Weights

In GCNs, the graph structure is typically represented using an adjacency matrix A, where $A_{ij} = 1$ if there is an edge between nodes i and j, and 0 otherwise. In weighted graphs, like the one used in our model, A_{ij} can take on values other than 0 and 1, representing the strength of the connection between nodes, we will later explain how this strength is calculated.

The adjacency matrix plays a crucial role in the message passing process. During the convolution operation, the edge weights in the adjacency matrix determine the strength of the influence that neighboring nodes have on each other. Nodes connected by edges with higher weights will have a stronger influence on each other's representations.

In our implementation, the 'GCNConv' layer from PyTorch Geometric is used, which efficiently implements this graph convolution operation:

This layer takes into account both the node features and the graph structure (encoded in the edge indices and edge attributes) to perform the graph convolution operation.

By using the graph structure in this way, GCNs are able to capture complex spatial relationships between nodes, making them powerful to capture the relationships between the sensor nodes activation in our network.

3.1.2 Long Short-Term Memory (LSTM) Networks

Long Short-Term Memory (LSTM) networks are a specialized form of Recurrent Neural Networks (RNNs) designed to capture long-term dependencies in sequential data. Introduced by Hochreiter and Schmidhuber in 1997 [1], LSTMs have becomeone of the most used model in time series analysis, natural language processing, and other domains involving sequential data.

The key innovation of LSTMs is their ability to selectively remember or forget information over long sequences, overcoming the vanishing gradient problem that plagues simple RNNs. This capability makes LSTMs particularly well-suited for tasks involving time-series data, where patterns may depend on events that occurred many time steps in the past.

LSTM Architecture

An LSTM unit consists of several components, each playing a crucial role in managing the flow of information:

1. Cell State (c_t): The internal memory of the LSTM, capable of maintaining information over long time periods.

2. Hidden State (h_t): The output of the LSTM at each time step, which can be used for predictions or as input to other layers.

3. Gates : Three gates control the flow of information in and out of the cell state:

- *Forget Gate (f_t):* Decides what information to discard from the cell state.

- *Input Gate (i_t):* Determines what new information to store in the cell state.

- *Output Gate (o_t):* Controls what information from the cell state should be output.

Mathematically, the operations of an LSTM cell can be described as follows:

1. *Forget Gate:*

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

2. *Input Gate:*

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

3. *Cell State Update:*

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

4. *Output Gate:*

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

5. *Hidden State Update:*

$$h_t = o_t \circ \tanh(c_t)$$

Where: - σ is the sigmoid function

- tanh is the hyperbolic tangent function

- $\circ$ denotes element-wise multiplication

- W_f, W_i, W_c, W_o are weight matrices

- b_f, b_i, b_c, b_o are bias vectors

In short, The forget gate determines what to remove from the previous cell state, the input gate decides what new information to add, and the output gate controls what parts of the cell state to output, and this is how LSTMs use gates to control the flow of information.

Advantages of LSTMs

In our research and study, we have found that LSTMs offer several key advantages in processing sequential data:

1. Long-term Dependencies : By using the cell state as a long-term memory, LSTMs can capture dependencies over many time steps.

2. Selective Memory : The gating mechanism allows the network to selectively remember or forget information, focusing on important parts of the input sequence.

3. Gradient Flow : The cell state provides a direct path for gradients to flow, this helps avoiding the vanishing gradient problem that is common in simple RNNs.

4. Flexibility : LSTMs can handle inputs of varying length and can be used for both many-to-one and many-to-many sequence tasks.

These characteristics make LSTMs particularly valuable in time-series forecasting tasks, and thus we chose to implement them to capture the temporal aspects of our graph network where capturing both short-term and long-term patterns is important for accurate predictions.

3.1.3 Customized LSTM Layer

In the context of sensor networks and time-series data, one challenge that often arises is dealing with irregular time intervals between data points. Standard LSTM models assume uniform time steps, which may not always be the case in real-world scenarios. To address this, we have implemented a customized LSTM layer that explicitly incorporates time delta information.

Motivation

The motivation behind this customization is to enable the model to adjust its predictions based on the actual time elapsed between observations. This is particularly important in scenarios where:

1. Data collection may be irregular or have gaps.

2. The rate of change in the observed phenomenon may vary with time.

3. The significance of past observations may decay at different rates depending on the time elapsed.

Structure of the Customized LSTM Layer

Our customized LSTM layer, implemented as the ‘CustomTemporalCell‘ class, extends the standard LSTM architecture by incorporating an additional input: the time delta (δ) between the current and previous observations.

The key components of this customized layer are:

1. Update Gate: Determines how much of the current input and previous hidden state to use.

2. Short-term Process : Handles immediate temporal dependencies.

3. Long-term Process: Manages longer-term temporal patterns.

4. Output Process: Combines short-term and long-term information for the final output.

The forward pass of this custom cell can be described mathematically as follows:

1. Update Gate:

$$u = \sigma(W_u[x_t, h_{t-1}, \delta_t] + b_u)$$

2. Short-term Process:

$$s_t = \tanh(W_s[x_t, h_{t-1}] + b_s)$$

$$h_t^s = u \circ h_{t-1} + (1 - u) \circ s_t$$

3. Long-term Process:

$$l_t = \tanh(W_l[x_t, h_{t-1}^l] + b_l)$$

$$u_l = \sigma(\delta_t)$$

$$h_t^l = u_l \circ h_{t-1}^l + (1 - u_l) \circ l_t$$

4. Output Process:

$$o_t = W_o[h_t^s, h_t^l] + b_o$$

Where: - x_t is the input at time t

- h_{t-1} is the previous hidden state

- δ_t is the time delta

- W_u, W_s, W_l, W_o are weight matrices

- b_u, b_s, b_l, b_o are bias vectors

- σ is the sigmoid function

- $\circ$ denotes element-wise multiplication

Key Features

1. Time-Aware Updates: The update gate incorporates the time delta, allowing the model to adjust its memory retention based on the time elapsed.

2. Dual-Time Scale Processing: By separating short-term and long-term processes, the model can capture both immediate changes and long-term trends.

3. Adaptive Long-term Memory: The long-term update is directly influenced by the time delta, allowing for more adaptive long-term memory retention.

4. Flexible Integration: This custom layer can be seamlessly integrated into larger network architectures, as demonstrated in the ‘Delta-GCN‘ model.

3.1.4 Client-Server Architecture in Federated Learning

Federated Learning (FL) is a machine learning paradigm that enables training models on distributed datasets without the need to centralize the data. This approach is particularly valuable in scenarios where data privacy is a concern or when data is naturally distributed across multiple locations or devices. The client-server architecture is a fundamental component of Federated Learning, facilitating collaborative learning while maintaining data locality.

Federated Learning, introduced by McMahan et al. in 2017 [2], aims to address the challenges of training machine learning models on decentralized data. The core idea is to bring the model to the data, rather than bringing all the data to the model. This paradigm shift offers several advantages:

1. Privacy Preservation: Sensitive data remains on local devices or servers. 2. Reduced Communication Overhead: Only model updates are transmitted, not raw data. 3. Scalability: Can handle a large number of participating clients. 4. Continuous Learning: Models can be updated as new data becomes available locally.

Client-Server Architecture

The client-server architecture in Federated Learning consists of two main components:

1. Clients: Individual entities (e.g., mobile devices, local servers, or sensor nodes) that possess local data and perform local model training.

2. Server: A central entity responsible for coordinating the learning process, aggregating model updates, and maintaining the global model.

The interaction between clients and the server follows a specific protocol:

1. Initialization: The server initializes the global model.

2. Client Selection: In each round, the server selects a subset of available clients to participate.

3. Model Distribution: The server sends the current global model to the selected clients.

4. Local Training: Each selected client trains the model on its local data for a specified number of epochs.

5. Update Aggregation: Clients send their model updates (not the raw data) back to the server.

6. Global Update: The server aggregates the updates from all participating clients to improve the global model.

7. Iteration: Steps 2-6 are repeated for a specified number of rounds or until convergence.

3.2 Data Preparation and Preprocessing

3.2.1 Raw Data Description

The raw motion data was collected by Mitsubishi Electric Research Labs (MERL) using a network of over 200 sensors. The dataset contains over 30 million raw motion records, spanning a full year and two floors of the research laboratory. This presents a challenge for behavior analysis, search, manipulation and visualization.of the data.

Sensor Device Description

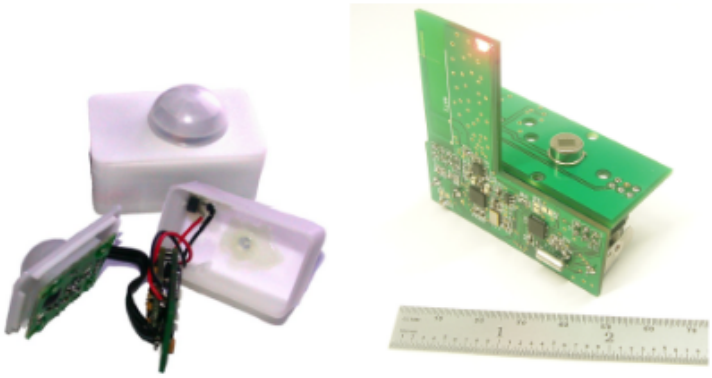


Figure 3.1: Passive infrared (PIR) motion detectors and MITes kit

The sensors used in this work are based on the MITes platform, integrated with a modified KC7783R sensor board. MITes is known for its affordability and adaptability, it's a versatile kit designed for pervasive computing research, particularly in environments like homes. It includes various sensors, such as motion, light, and temperature sensors, capable

of operating simultaneously, making it suitable for comprehensive motion data collection. The MITes-based node utilized in this work employs an older network protocol without checksum verification, leading to potential packet loss, duplication, or corruption. Although efforts were made to filter out erroneous data, we still end up with some loss in data, which should be considered in the analysis. The sensor boards are passive infrared (PIR) motion detectors that sense infrared light emitted by warm objects, modified to enhance responsiveness by reducing the adaptation rate from minutes to seconds, with a typical inter-detection time of around 1.5 seconds. While these sensors are effective for capturing rapid motion events, they have limitations, including sensitivity to temperature changes, shocks, and vibrations, and are primarily designed for indoor use. These characteristics, while enabling detailed motion detection, introduce potential variability in the collected data, which must be considered during analysis.

Data Files Description

The dataset is divided into several files:

Raw motion data: It is captured by the motion detector sensors that are ceiling mounted at about two meter intervals covering the space in the hallways and meeting rooms. The file that contains the data is organized like:

470 01179980510828 01179980511853 1.0
469 01179980512169 01179980513193 1.0
467 01179980513580 01179980514609 1.0
468 01179980514573 01179980515598 1.0

Where the first element is the sensor identification number. The second and third numbers are the timestamps of the beginning of the event. The fourth number is just a place holder value.

Map data: Each sensor ID corresponds to a unique sensor. The sensors IDs are associated with physical space in the floor, each sensor is represented in a row,with the sensor ID followed by eight coordinates that specify the four corners of a quadrilateral in meters:

sid,x1,y1,x2,y2,x3,y3,x4,y4
214,-13.3,23.1,-13.3,25.3,-15.5,25.3,-15.5,23.1
222,-13.3,20.9,-13.3,23.1,-15.5,23.1,-15.5,20.9
256,-15.5,8.3,-15.5,10.5,-17.7,10.5,-17.7,8.3

257,-13.3,8.3,-13.3,10.5,-15.5,10.5,-15.5,8.3

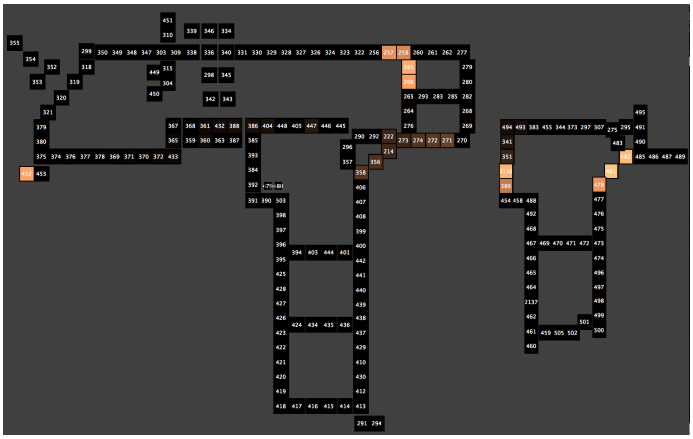


Figure 3.2: Floor 8 and Floor 7 map with sensor IDs

Time data: Timestamps are the number of milliseconds since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time, without including leap seconds. That is, 1000*time(); in C, for example:

Mon Jul 2 14:54:42 EDT 2007
1183402482
1183402482861

3.2.2 Dataset Construction:

Going from our data in .txt file, we will no showcase how we could transform the simple motion data into a set of temporal graph datasets so that we can train and test the model on them, we can generalize the process in the figure 3.3

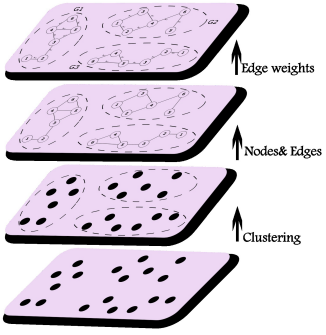


Figure 3.3: Graph Construction process

we first started by cleaning up the data in the text file and then inverting it so we could have rows of activity data with each column representing a sensor node and the row representing the floor graph.

Clustering Process and Algorithm

Once we had our data cleaned and organized, we took the spatial coordinates of every sensor node and plotted them, later, using the global adjacency matrix that we created, we were able to create clusters of nodes that we consider as our new graph datasets, with varying topologies and varying sizes. the clustering process can be summarized in the following algorithm:

Algorithm 2 Spectral Clustering Algorithm

Require: A (Weighted adjacency matrix), sensor IDs, k (Cluster size)
Ensure: List of clusters

```
Initialize sensor_ids as the list of available sensors
Initialize clusters ← []
while |sensor_ids| ≥ k do
    Select the first sensor as the seed
    Remove the seed from sensor_ids
    Initialize a new cluster with the seed
    while |cluster| < k do
        Find neighbors of all sensors in the cluster
        Sort neighbors by their connection strength
        for each neighbor in sorted list do
            if neighbor is not already in the cluster then
                Add neighbor to the cluster
                Remove neighbor from sensor_ids
            end if
            if |cluster| = k then
                break
            end if
        end for
        if no new sensors are added after 3 attempts then
            break
        end if
    end while
    Add the cluster to clusters
end whilereturn clusters
```

Graph Datasets Construction

We developed a custom dataset for graph-based time series analysis. The core of this process is a ConstantGraphDataset class we created, which inherits from PyTorch's Dataset class. This class takes two main inputs: a DataFrame with node features and temporal data, and another DataFrame representing the graph's adjacency matrix.

We designed the class to extract node features from the first DataFrame, converting them into a tensor suitable for graph convolutions. The temporal information, crucial for our analysis, is stored in a 'delta' column, which we also convert to a tensor.

To represent the graph structure, We process the adjacency matrix, creating tensors for edge indices and edge attributes. This approach allows me to capture both the connections between nodes and the strength of these connections.

For each data point, We create a Data object that combines node features, edge information, and temporal data. To handle batching, We implemented a custom collate function that efficiently combines multiple graphs.

We split the dataset into training, validation, and test sets using a 70-20-10 ratio. This split ensures We have enough data for training while reserving portions for validation and final testing. We also calculate class weights to address potential imbalances in our dataset.

Finally, We create DataLoader objects for each set, incorporating our custom collate function. These loaders use a batch size that We optimized for our available computational resources.

This dataset structure allows our model to process both the spatial (graph) and temporal aspects of the data simultaneously, which is crucial for our research in graph-based time series analysis.

3.3 Proposed Delta-GCN Model and FL Architecture

3.3.1 Architecture Overview

The proposed Delta-GCN model combines the strengths of Graph Convolutional Networks (GCNs) and a customized Long Short-Term Memory (LSTM) architecture to effectively capture both spatial and temporal dependencies in sensor networks. and to further enhanced the implementation and generalize it on different topologies, we integrated this model into a Federated Learning (FL) framework, allowing for distributed training across multiple networks (clients) while preserving data privacy. The architecture consists of three main components:

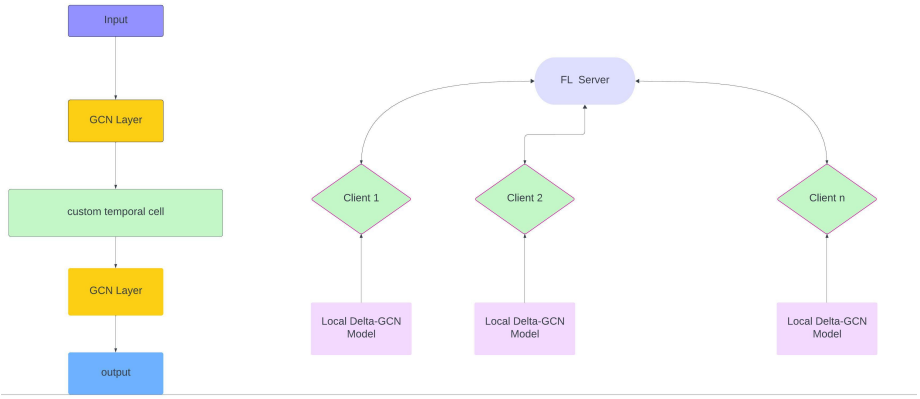


Figure 3.4: High-level architecture of the Delta-GCN model in a federated learning setting

-GCN layers for capturing spatial correlations

-A custom temporal cell for processing time-dependent data with irregular intervals

-A federated learning framework for distributed training and privacy preservation

3.3.2 GCN Component

The GCN component of our model is used for capturing the spatial correlations between sensor nodes in the network. It leverages the graph structure of the sensor network to aggregate information from neighboring nodes, thereby learning rich representations that encode both node features and network topology.

In our implementation, we use two GCN layers: an initial layer to process the input features and a final layer to produce the output. The GCN layer is defined using the GCNConv class from the PyTorch Geometric library:

```
self.initial_gcn = GCNConv(input_size, hidden_size)
self.final_gcn = GCNConv(hidden_size, output_size)
```

To enhance the model's performance, we incorporate batch normalization and dropout:

```
self.bn1 = nn.BatchNorm1d(hidden_size)
self.bn2 = nn.BatchNorm1d(hidden_size)
self.dropout = nn.Dropout(0.2)
```

These techniques help in regularizing the model and improving its generalization on other graph structures and prevent overfitting.

3.3.3 Customized LSTM Component

The core of our temporal processing is the CustomTemporalCell, a modified LSTM architecture designed to handle irregular time intervals between sensor readings. This custom cell is crucial for dealing with the variable delta times often encountered in real-world sensor networks. The CustomTemporalCell is defined as follows:

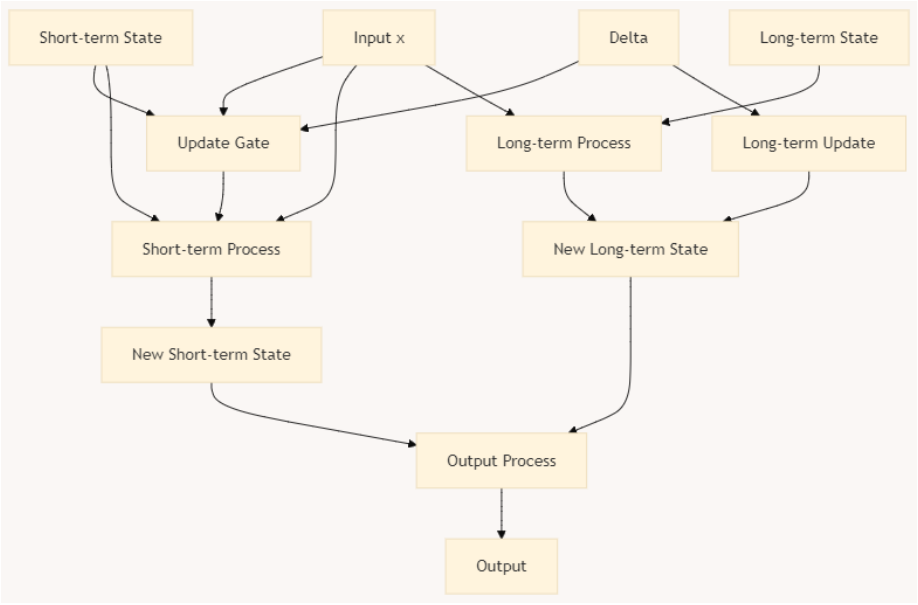


Figure 3.5: Structure of the CustomTemporalCell

Algorithm 3 Custom Temporal Cell: Processing Time-Dependent Data

function TemporalCell(input, short-term memory, long-term memory, time difference)
    Combine current input with short-term memory and time information
    Decide how much of the short-term memory to keep or update
    Create a new short-term memory proposal
    Update short-term memory based on the decision and proposal
    Determine long-term update intensity based on time difference
    Create a new long-term memory proposal
    Update long-term memory based on update intensity and proposal
    Generate output using both updated short-term and long-term memories
    return output, updated short-term memory, updated long-term memory
end function

The key features of this custom cell are:

Update Gate: This gate determines how much of the current input and previous

short-term state should be used to update the new short-term state. It takes into account the time delta, allowing the model to adjust its updates based on the time elapsed since the last observation.

Short-term and Long-term States: The cell maintains separate short-term and long-term states. The short-term state is updated at every time step, while the long-term state is updated based on the time delta. This dual-state approach allows the model to capture both immediate temporal dependencies and longer-term trends.

Delta-based Long-term Update: The long-term state is updated using a sigmoid function of the time delta. This ensures that the long-term state is updated more significantly when there's a larger time gap between observations.

Output Process: The final output is computed using both the short-term and long-term states, allowing the model to leverage both immediate and historical information.

The mathematical formulation of the CustomTemporalCell can be expressed as follows:

$$
\begin{aligned}
u_t &= \sigma(W_u[x_t, h_{t-1}, \delta_t] + b_u) \\
\tilde{h}_t &= \tanh(W_h[x_t, h_{t-1}] + b_h) \\
h_t &= u_t \odot h_{t-1} + (1 - u_t) \odot \tilde{h}_t \\
u_t^l &= \sigma(\delta_t) \\
\tilde{l}_t &= \tanh(W_l[x_t, l_{t-1}] + b_l) \\
l_t &= u_t^l l_{t-1} + (1 - u_t^l) \odot \tilde{l}_t \\
o_t &= W_o[h_t, l_t] + b_o
\end{aligned}
$$

where x_t is the input at time t, h_t is the short-term state, l_t is the long-term state, δ_t is the time delta, u_t is the update gate, u_t^l is the long-term update gate, and o_t is the output.

3.3.4 Federated Training Process

The Federated Learning (FL) framework is implemented to enable distributed training across multiple clients while preserving data privacy. we used this architecture for scenarios where sensor data is distributed across different locations or organizations, and centralized data collection is not feasible due to privacy concerns or regulatory requirements, which is the case for real life data.

The FL process in our implementation follows these steps:

1-Model Initialization: The server initializes the global Delta-GCN model.

2-Client Selection: In each round, the server selects a subset of available clients to participate in the training process.

3-Local Training: Each selected client receives the current global model and performs local training using its own data:

Algorithm 4 Fit Function for Federated Learning

```
1: function FIT(parameters, config)
2:     SetParameters(net, parameters)
3:     Train(net, trainloader, epochs = 1)
4:     return GetParameters(net), len(trainloader), {}
5: end function
```

The fit method in the Client class represents this local training process.

4-Model Update Aggregation: After local training, clients send their model updates back to the server. The server aggregates these updates to create a new global model:

```
def aggregate(self, results):
```

In this step we are averaging the model parameters across all participating clients (FedAvg algorithm)

5-Model Distribution: The updated global model is then distributed back to the clients for the next round of training.

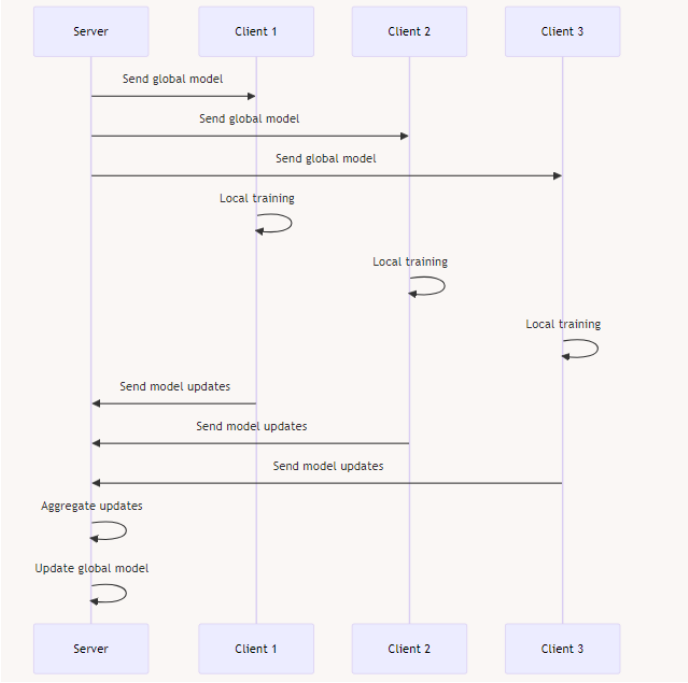


Figure 3.6: Federated Learning Process

3.3.5 Data Imputation Using Model Predictions

Our Delta-GCN model had a function for data imputation, addressing the common challenge of missing data in sensor networks, but also, saving energy by turning off the sensors and predicting their readings We employ a semi-supervised approach where the model learns to predict missing values while simultaneously being trained on the available data.

The imputation process involves the following steps:

Mask Creation: We create a mask for each batch of data, randomly setting a portion of the input values to a placeholder value (in our case, -1):

Algorithm 5 Create Mask for Batch

```
function CreateMaskForBatch(batch_size, num_nodes, mask_ratio)
    mask ← ZeroTensor(batch_size × num_nodes)
    for i = 0 to batch_size − 1 do
        start ← We × num_nodes
        end ← (i + 1) × num_nodes
        mask[start : end] ← Bernoulli(FullTensor(num_nodes, mask_ratio))
    end for
    return mask
end function
```

Forward Pass: The masked data is passed through the Delta-GCN model, which generates predictions for all nodes, including the masked ones.

Loss Calculation: We calculate the loss only for the masked values, comparing the model's predictions to the true values, this significantly helps the model to adjust and focus of predicting the missing values accuratly:

Algorithm 6 Masked Loss: Focusing on Important Predictions

```
function MaskedLoss(predictions, true values, mask, positive weight)
    Select only the predicted values where the mask is applied
    Select only the true values where the mask is applied
    Calculate the weighted error between these predictions and true values
    return the calculated error (loss)
end function
```

Backpropagation: The calculated loss is used to update the model parameters, improving its ability to predict missing values.

Evaluation: We evaluate the model's imputation performance using various metrics:

Algorithm 7 Evaluate Imputation

```
function EvaluateImputation(model, data, mask, device)
    true_values ← data.x[mask]
    imputed_values ← model(data)[mask]
    f2_score ← FBetaScore(true_values, imputed_values, β = 2)
    precision ← PrecisionScore(true_values, imputed_values)
    recall ← RecallScore(true_values, imputed_values)
    accuracy ← Mean(true_values = imputed_values)
    f_score ← F1Score(true_values, imputed_values)
    return f2_score, precision, recall, accuracy, f_score
end function
```

These metrics provide a comprehensive view of the model's imputation capabilities, considering both the accuracy and the balance between precision and recall. we integrated the imputation process into the training loop, allowing the model to simultaneously learn from available data and improve its imputation abilities on the masked one:

Algorithm 8 Training Loop: Learning from Partially Masked Data

```
for each training round do
    for each batch of data do
        Create a random mask to simulate missing data
        Apply the mask to the input data
        Pass the masked data through the model
        Compare model's predictions with actual values at masked positions
        Calculate how wrong the predictions are (loss)
        Use this error to improve the model
    end for
end for
```

This approach has several advantages:

Adaptive Learning: The model learns to impute values in the context of the entire sensor network, considering both spatial and temporal dependencies.

Robustness: By training with partially masked data, the model becomes more robust to missing values in real-world scenarios.

Continuous Improvement: The imputation capability of the model improves alongside its primary task performance throughout the training process.

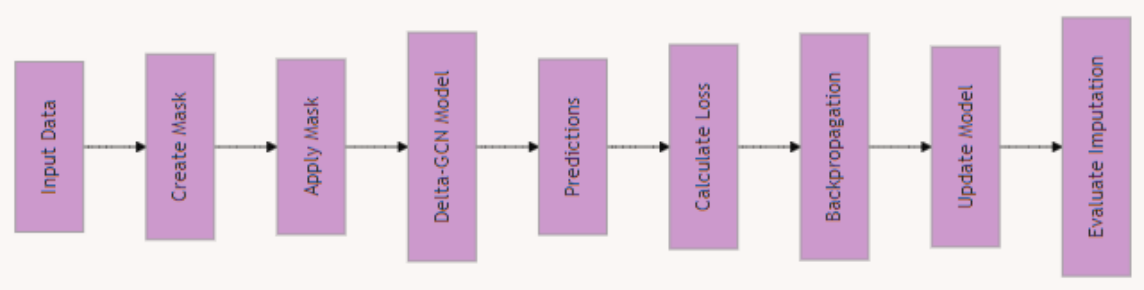


Figure 3.7: Data Imputation Process

In conclusion, our Delta-GCN model, combined with federated learning and integrated data imputation, presents a comprehensive solution for handling spatiotemporal data in distributed sensor networks. It effectively captures both spatial and temporal dependencies, preserves data privacy through federated learning, and addresses the challenge of missing data through its imputation, which we mainly designed for energy effeciency in the network.

3.4 General Algorithm

We can represent the whole training and testing of our Delta-GCN model in a federated learning setting with this algorithms:

Algorithm 9 Federated Learning for Delta-GCN

Input: Clients $C_1, \ldots, C_N$, each with local dataset D_i
Initialize: Global Delta-GCN model M_g
Initialize: Number of rounds R, local epochs E
for each round $r = 1$ to R **do**
  for each client C_i in parallel **do**
    $M_i \leftarrow M_g$ ▷ Download global model
    $D_i^{train}, D_i^{val} \leftarrow \text{SplitDataset}(D_i)$
    for epoch $e = 1$ to E **do**
      for batch b in D_i^{train} **do**
        mask $\leftarrow \text{CreateMask}(b, \text{mask\_ratio})$
        b.x[mask] $\leftarrow 0.2$ ▷ Mask input
        out $\leftarrow M_i(b)$
        loss $\leftarrow \text{MaskedLoss}(\text{out}, b.\text{x}, \text{mask}, w_+)$
        $\text{UpdateModel}(M_i, \text{loss})$
      end for
    end for
    $\text{metrics}_i \leftarrow \text{Evaluate}(M_i, D_i^{val})$

Send M_i and metrics_i to server
end for
$M_g \leftarrow \text{AggregateModels}(M_1, \ldots, M_N)$
$\text{metrics}_g \leftarrow \text{AggregateMetrics}(\text{metrics}_1, \ldots, \text{metrics}_N)$
Update learning rate based on metrics_g
end for
return M_g

Where the M_g and M_i models are :

Algorithm 10 Delta-GCN Model

Input: Graph $G = (V, E)$, node features X, edge attributes A, time deltas Δ
Initialize:
GCN layers: $\text{GCN}_{\text{initial}}$, $\text{GCN}_{\text{final}}$
Temporal Cell: TC with short-term state h_s and long-term state h_l
function DELTA-GCN(X, E, A, Δ)
$H_0 \leftarrow \text{GCN}_{\text{initial}}(X, E, A)$ ▷ Initial GCN layer
$H_0 \leftarrow \text{ReLU}(\text{BatchNorm}(H_0))$
$H_0 \leftarrow \text{Dropout}(H_0)$
$h_s, h_l \leftarrow \mathbf{0}$ ▷ Initialize temporal states
for each node v in V **do**
$h_v, h_s, h_l \leftarrow \text{TC}(H_0[v], h_s, h_l, \Delta[v])$
$H_1[v] \leftarrow h_v$
end for
$H_1 \leftarrow \text{BatchNorm}(H_1)$
$Y \leftarrow \text{GCN}_{\text{final}}(H_1, E, A)$ ▷ Final GCN layer
$Y \leftarrow \text{Sigmoid}(Y)$
return Y
end function
function TEMPORALCELL(x, h_s, h_l, δ)
$z \leftarrow \sigma(W_z[x, h_s, \delta] + b_z)$ ▷ Update gate
$\tilde{h}_s \leftarrow \tanh(W_s[x, h_s] + b_s)$ ▷ Short-term candidate
$h_s \leftarrow z \odot h_s + (1 - z) \odot \tilde{h}_s$ ▷ Short-term update
$z_l \leftarrow \sigma(\delta)$ ▷ Long-term update gate
$\tilde{h}_l \leftarrow \tanh(W_l[x, h_l] + b_l)$ ▷ Long-term candidate
$h_l \leftarrow z_l \odot h_l + (1 - z_l) \odot \tilde{h}_l$ ▷ Long-term update
$y \leftarrow W_o[h_s, h_l] + b_o$ ▷ Output
return y, h_s, h_l
end function

3.5 Evaluation Metrics and Comparison

In this section, we discuss the key evaluation metrics used to assess the performance of the proposed model, which focuses on energy-saving in wireless sensor networks. These metrics are chosen based on their relevance to both the prediction task and the efficiency of the network. We also outline the methods used for comparing the proposed model against state-of-the-art solutions.

3.5.1 F2 Score

The F2 Score is a variation of the more commonly used F1 Score, placing higher importance on recall compared to precision. This metric is particularly useful in applications where false negatives are more critical than false positives. In the context of our wireless sensor network (WSN) model, the F2 Score is vital for ensuring that the system detects energy consumption patterns or node failures (the positive class) effectively, minimizing the risk of undetected issues.

The F2 score is calculated as:

$$F2 = \frac{(1 + 2^2) \cdot \text{precision} \cdot \text{recall}}{2^2 \cdot \text{precision} + \text{recall}}$$

Where:

- **Precision** is the ratio of true positives to the sum of true positives and false positives.
- **Recall** is the ratio of true positives to the sum of true positives and false negatives.

Since recall is emphasized over precision in the F2 score, this metric helps ensure that the system is more forgiving of false positives but sensitive to missing true positives, which is critical for energy management in sensor networks.

3.5.2 Recall and Precision

Recall is a measure of how well the model can identify positive instances, i.e., how many of the actual positive cases were correctly predicted by the model. In the context of WSN, a high recall ensures that the model correctly identifies nodes with anomalous energy consumption or any other important event.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Precision, on the other hand, measures the accuracy of positive predictions, i.e., how many of the predicted positive instances are truly positive. In our scenario, high precision ensures that when the model flags a node for further investigation (e.g., for high energy consumption), it is likely to be correct, thus avoiding unnecessary energy-draining actions such as unnecessary communication or maintenance.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

The balance between precision and recall is important in WSNs, as we want to avoid both false alarms (low precision) and missed detections (low recall). By focusing on both metrics, we can ensure the model makes reliable predictions for energy-saving strategies.

3.5.3 Energy Consumption

Energy consumption is a critical metric for wireless sensor networks, especially when dealing with long-term deployments in environments where recharging or replacing batteries is impractical. In this project, energy consumption refers to the total amount of energy used by the sensor nodes during the data collection, transmission, and processing phases.

The total energy consumption of a network can be modeled as:

$$E_{\text{total}} = \sum_{i=1}^{N} E_{i,\text{transmit}} + E_{i,\text{receive}} + E_{i,\text{process}}$$

Where:

- N is the number of sensor nodes in the network,
- $E_{i,\text{transmit}}$ is the energy consumed by node i during transmission,
- $E_{i,\text{receive}}$ is the energy consumed by node i during reception,
- $E_{i,\text{process}}$ is the energy consumed by node i during data processing.

The proposed model aims to minimize energy consumption by reducing unnecessary transmissions and computations, particularly in the federated learning context. The GCN-LSTM model works by capturing spatial-temporal relationships, allowing for optimized data processing that extends the network's operational period.

3.5.4 Network Lifetime

Network lifetime refers to the duration for which the wireless sensor network remains functional before critical nodes fail due to battery depletion. This is an essential metric for

evaluating the effectiveness of energy-saving strategies. A longer network lifetime indicates that the energy-efficient mechanisms are working, allowing the network to operate longer without the need for intervention.

The network lifetime can be defined as the time until the first node depletes its battery or the time until a specific percentage of nodes fail, depending on the application. A simple formula for network lifetime is:

$$T_{\text{lifetime}} = \frac{E_{\text{initial}}}{E_{\text{avg}}}$$

Where:

- E_{initial} is the initial energy of a sensor node,
- E_{avg} is the average energy consumption per unit time.

By comparing network lifetime across models, we can assess whether the proposed energy-saving strategies successfully extend the operational duration of the sensor network.

3.5.5 Comparison with State-of-the-Art Solutions

Our approach is built upon a comprehensive review of existing methods, where we identified key shortcomings in previous works. Most state-of-the-art solutions treat the problem either as a graph optimization task primarily addressing the spatial dependencies between sensors or as a time series optimization problem aimed at capturing the temporal relationships between events. However, these methods often overlook the irregular nature of time dependencies in real-world Wireless Sensor Networks (WSNs). In response, we propose an innovative solution that integrates both spatial and temporal aspects through a novel LSTM-based model designed specifically to account for irregular time intervals in sensor data.

Moreover, many high-performing approaches in the literature demand substantial computational resources for training and deployment, making them unsuitable for the resource-constrained environment of WSNs. Our Delta-GCN model, by contrast, is computationally efficient and well-suited for low-power devices typical in WSN deployments.

Another key innovation in our approach is the use of a decentralized training framework. Unlike centralized methods, which require transferring data to a single server (introducing security risks and communication overhead) our decentralized approach aligns with real-world applications where data is distributed across multiple nodes. This not only enhances data security by keeping sensitive information local but also significantly reduces energy

consumption by minimizing communication overhead. By eliminating the need to transfer large volumes of data to a central server, we achieve both energy efficiency and scalability, making our approach highly practical for WSNs.

3.6 Conclusion

In this respect, our Delta-GCN model introduces a comprehensive solution to both of these challenges: the modeling of spatial and temporal dependencies in WSNs. By jointly leveraging the strengths of GCNs and a custom LSTM architecture designed for dealing with irregular time intervals, our approach substantially improves real-world sensor data modeling. The contribution in this manuscript makes the model more applicable to real-world scenarios, integrating it into an FL framework, hence enabling decentralized model training over numerous sensor networks while ensuring data privacy and minimizing the communications overhead.

Our innovations do not stop at addressing missing data with a semi-supervised imputation approach; we also underline energy efficiency as one of the most important factors when it comes to WSN environments. The Delta-GCN model accomplishes a good balance between computational efficiency and effective learning by reducing the need to keep sensors constantly active and minimizing data transfer, hence providing a practical and scalable solution for modern deployments of WSNs. The contribution of our work has advanced the state of the art for energy-efficient systems in WSNs and sets the ground for further research in resource use optimization within similar distributed environments.

Chapter 4

Implementation of The Solution

4.1 Technologies and Tools

For the development of our system, we opted for a set of development languages, technologies, and frameworks as follows:

4.1.1 Python

Python is an object-oriented, interpreted, multi-paradigm, and cross-platform programming language. It features dynamic typing, automatic memory management via garbage collection, and an exception management system. Thanks to its high-level interactivity and extensive library ecosystem, it is an attractive choice for algorithm development and exploratory data analysis.

Why Python for this type of project? Several reasons behind Python's structure make it the most suitable language for machine learning (ML) algorithms and data exploitation.

First, the existence of numerous Python libraries and frameworks specific to machine learning simplifies the process and reduces development time. Also, Python's simple syntax and readability allow for quick testing of complex algorithms, making the language accessible to non-programmers. Finally, the simple syntax facilitates collaboration or project transfer between developers. Python also boasts a large, active community of developers who are happy to offer their help and support, which can be invaluable for complex projects.

PyTorch:

A popular open-source machine learning framework developed by Facebook's AI Research lab. PyTorch is widely used for deep learning applications due to its dynamic computation graph, ease of debugging, and integration with Python libraries.

PyTorch Geometric:

A library built on top of PyTorch, specifically designed for handling graph-based machine learning models. It provides tools for deep learning on graphs and other irregularly structured data.

Numpy:

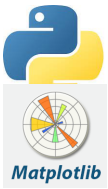
The fundamental package for scientific computing with Python. This library provides multiple functions for creating or saving arrays and manipulating vectors, matrices, and polynomials.

Pandas:

A Python library for data manipulation and analysis. It particularly offers data structures and operations for manipulating numerical arrays and time series.

Matplotlib.pyplot:

A collection of command-style functions that make Matplotlib work like MATLAB. Each Pyplot function makes a change to a figure: for example, creating a figure, creating a plot area within a figure, plotting lines in a plot area, decorating the figure with labels, etc. Pyplot is mainly used for interactive graphs and simple cases of automatic graph generation.



4.1.2 Google Colaboratory

Google Colab or Colaboratory is a cloud service (free) offered by Google, based on Jupyter Notebook, designed for training and research in machine learning. This platform allows training machine learning models directly in the cloud, so there is no need to install anything on our computer, except for a browser.

4.1.3 Flower

Flower is a framework for building federated learning systems. It provides the necessary tools to enable collaboration between different machines while keeping data decentralized. It allows for easy experimentation and implementation of federated learning models, making it an essential tool in distributed machine learning scenarios.

4.1.4 Visual Studio Code

Visual Studio Code (VS Code) is a free and open-source code editor developed by Microsoft. It supports numerous programming languages, including Python, and offers features such as code debugging, Git control, and extensions that enhance the development workflow. It is widely used by developers for Python development due to its ease of use and extensive functionality.

4.1.5 Google Drive

Google Drive is a cloud storage service offered by Google that allows users to store files online, sync them across devices, and share them with others. It is widely used for collaborative work, making it a convenient tool for storing project files, datasets, and code, and for sharing them with team members and supervisors.

4.1.6 Git and GitHub

Git: A decentralized version control software. It is free software created by Linus Torvalds, the author of the Linux kernel, and distributed under the GNU General Public License version 2. As of 2016, it is the most popular version control software, used by more than twelve million people.

Github: A web hosting and software development management service that uses Git. This site is developed in Ruby on Rails. It also provides access control and collaboration features such as bug tracking, feature requests, task management, and a wiki for each project.

4.2 Solution Implementation

4.2.1 Class Diagram

We can represent the classes used in this implementation with the following class diagram:

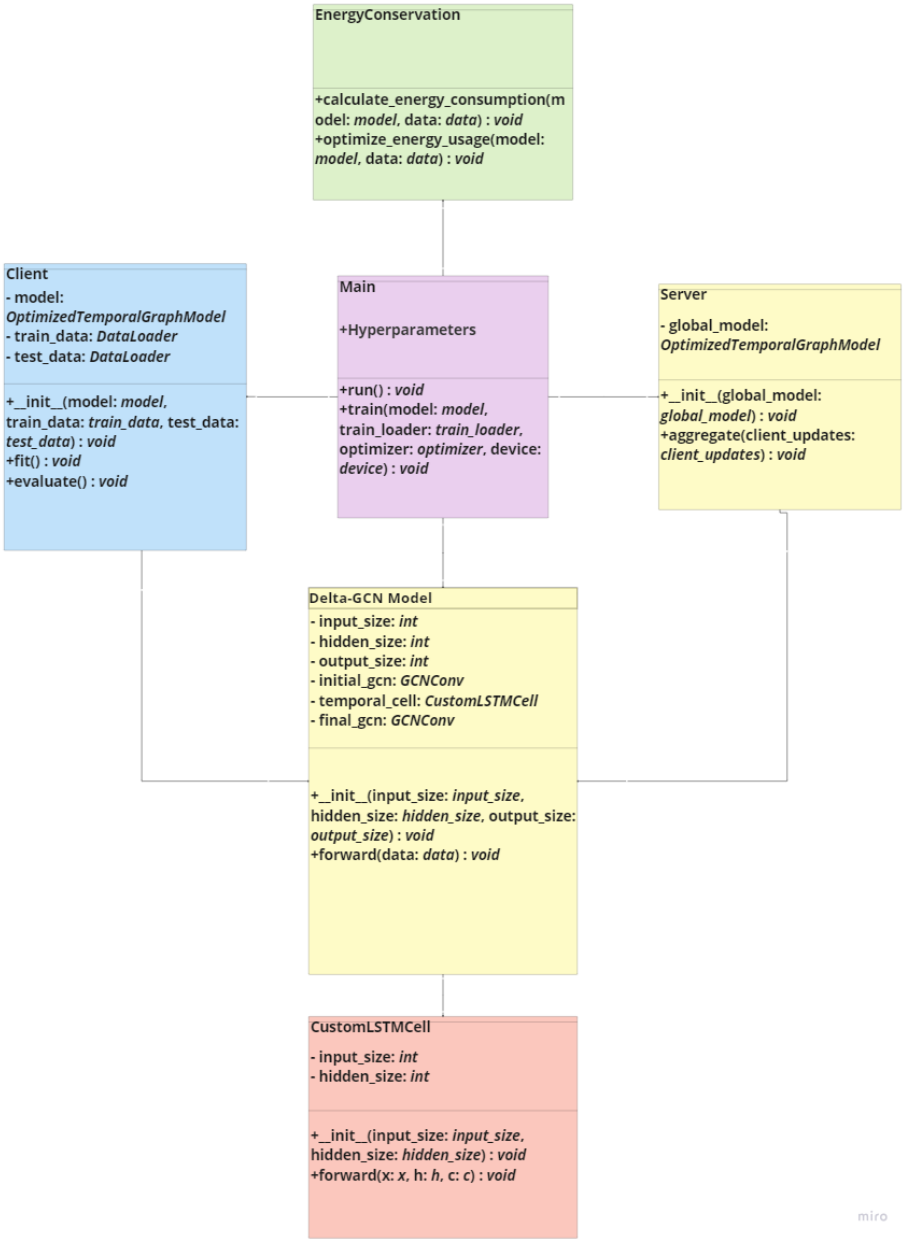


Figure 4.1: Class diagram

4.2.2 Class Description

Here we will explain the class functions:

Class Name	Description
GraphDataset	Manages graph data preprocessing by converting input DataFrames and graph structures into PyTorch tensors and PyTorch Geometric Data objects for GNN training. Handles node features, edge information, and temporal deltas.
CustomTemporalCell	Neural network module implementing a temporal processing unit with short-term and long-term memory states, incorporating temporal gaps (delta) for time-aware graph processing.
Delta-GCN	The main graph neural network architecture combining GCN layers with temporal processing. It processes graph data through an initial GCN layer, feeds the results through a CustomTemporalCell for temporal processing, and produces final predictions via a second GCN layer. Includes dropout (0.2), batch normalization, and sigmoid activation for output probabilities. Designed for handling temporal graph data with varying time gaps.
Client	Implements federated learning client functionality, managing local model training and parameter synchronization within the distributed learning framework.
Server	Implements federated learning Server functionality, Initializing the model and aggregating the model parametars and also sending the model updates to the clients.
DataManager	Handles data pipeline including dataset splitting, dataloader creation, mask generation, and evaluation metrics calculation for model training and assessment.
FederatedLearning-Manager	Coordinates federated learning process by managing server-client interactions, training rounds, and model parameter aggregation across distributed clients.

Table 4.1: Condensed Class Documentation

4.2.3 Federated Learning Setup

The Federated Learning (FL) framework is implemented using the Flower library, which provides a robust infrastructure for federated learning experiments. Our setup consists of multiple clients (sensor nodes) and a central server, facilitating distributed training while

preserving data privacy.

Client-side Training:

Each client in our federated learning setup is represented by an instance of the *FlowerClient* class, which inherits from Flower's *NumPyClient*:

```
class FlowerClient(NumPyClient):
    def __init__(self, net, trainloader, valloader):
        self.net = net
        self.trainloader = trainloader
        self.valloader = valloader

    def get_parameters(self, config):
        return get_parameters(self.net)

    def fit(self, parameters, config):
        set_parameters(self.net, parameters)
        train(self.net, self.trainloader, epochs=config['local_epochs'])
        return get_parameters(self.net), len(self.trainloader), {}

    def evaluate(self, parameters, config):
        set_parameters(self.net, parameters)
        loss, accuracy = test(self.net, self.valloader)
        return float(loss), len(self.valloader), {"accuracy": float(accuracy)}
```

The *fit* method is responsible for local training on each client. It receives the global model parameters, sets them to the local model, performs training for a specified number of epochs, and returns the updated parameters along with the size of the local dataset.

The local training process is defined in the train function:

```
def train(net, trainloader, epochs, device, optimizer, criterion):
net.train()
for epoch in range(epochs):
    for batch in trainloader:
        batch = batch.to(device)
        optimizer.zero_grad()
        out = net(batch)
        loss = criterion(out, batch.y)
        loss.backward()
```

```
            optimizer.step()
```

Server-side Aggregation:

The server aggregates model updates from clients using the Federated Averaging (FedAvg) algorithm. This is implemented in the *FedAvg* strategy provided by Flower:

```
strategy = fl.server.strategy.FedAvg(
 fraction_fit=0.1,
fraction_evaluate=0.2,
min_fit_clients=2,
min_evaluate_clients=2,
min_available_clients=2,
)
```

The FedAvg strategy performs weighted averaging of client updates, where the weights are proportional to the number of samples in each client's local dataset.

To start the federated learning process, we initialize the server and client applications:

```
server = ServerApp(server_fn=server_fn)
client = ClientApp(client_fn=client_fn)

def server_fn(context: Context) -> ServerAppComponents:
    config = ServerConfig(num_rounds=NUM_ROUNDS)
    return ServerAppComponents(strategy=strategy, config=config)

def client_fn(context: Context) -> Client:
    net = Delta-GCN().to(DEVICE)
    partition_id = context.node_config["partition-id"]
    trainloader, valloader, _ = load_datasets(partition_id=partition_id)
    return FlowerClient(net, trainloader, valloader).to_client()
```

The $server_fn$ and $client_fn$ functions define the server and client configurations, respectively. The NUM_ROUNDS parameter determines the number of federated learning rounds to be performed.

4.2.4 Delta-GCN Model Implementation

The Delta-GCN model combines Graph Convolutional Network (GCN) layers for spatial correlation learning with a custom temporal cell for processing time-dependent data. The

model is implemented as follows:

```
class Delta−GCN(nn.Module):
def ___init___(self, input_size, hidden_size, output_size, dropout_rate):
    super().___init___()
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.output_size = output_size

    self.dropout = nn.Dropout(dropout_rate)

    self.initial_gcn = GCNConv(input_size, hidden_size)
    self.temporal_cell = CustomTemporalCell(hidden_size, hidden_size)
    self.final_gcn = GCNConv(hidden_size, output_size)

    self.bn1 = nn.BatchNorm1d(hidden_size)
    self.bn2 = nn.BatchNorm1d(hidden_size)

def forward(self, data):
    x, edge_index, edge_attr, delta = data.x, data.edge_index, data.edge_attr,
        data.delta
    batch_size = data.num_graphs

    x = self.dropout(F.relu(self.bn1(self.initial_gcn(x, edge_index, edge_attr))))

    short_term_state = torch.zeros(batch_size, self.hidden_size, device=x.device)
    long_term_state = torch.zeros(batch_size, self.hidden_size, device=x.device)

    x = x.view(batch_size, −1, self.hidden_size)
    outputs = []
    for i in range(x.size(1)):
        output, short_term_state, long_term_state = self.temporal_cell(
            x[:, i, :], short_term_state, long_term_state, delta
        )
        outputs.append(output)
    x = torch.stack(outputs, dim=1)
    x = x.view(−1, self.hidden_size)

    x = self.bn2(x)
```

```
        x = self.final_gcn(x, edge_index, edge_attr)
        return torch.sigmoid(x)
```

The GCN component is implemented using the *GCNConv* layer from PyTorch Geometric. The temporal component is a custom LSTM-like cell designed to handle irregular time intervals:

```
class CustomTemporalCell(nn.Module):
def __init__(self, input_size, hidden_size):
    super().__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size

    self.update_gate = nn.Linear(input_size + hidden_size + 1, hidden_size)
    self.short_term_process = nn.Linear(input_size + hidden_size, hidden_size)
    self.long_term_process = nn.Linear(input_size + hidden_size, hidden_size)
    self.output_process = nn.Linear(2 * hidden_size, hidden_size)

def forward(self, x, short_term_state, long_term_state, delta):
    combined = torch.cat([x, short_term_state, delta.unsqueeze(-1)], dim=-1)
    update = torch.sigmoid(self.update_gate(combined))

    short_term_candidate = self.short_term_process(torch.cat([x,
        short_term_state], dim=-1))
    short_term_state = update * short_term_state + (1 - update) *
        short_term_candidate

    long_term_update = torch.sigmoid(delta).unsqueeze(-1)
    long_term_candidate = self.long_term_process(torch.cat([x, long_term_state],
         dim=-1))
    long_term_state = long_term_update * long_term_state + (1 -
        long_term_update) * long_term_candidate

    output = self.output_process(torch.cat([short_term_state, long_term_state],
        dim=-1))

    return output, short_term_state, long_term_state
```

Hyperparameters and Configuration:

The model's hyperparameters are defined as follows:

```
    HYPERPARAMETERS = {
    'input_size': 1,
    'hidden_size': 256,
    'output_size': 1,
    'dropout_rate': 0.2,
    'learning_rate': 1e-4,
    'weight_decay': 1e-5,
    'batch_size': 32000,
    'num_epochs': 100,
    'mask_ratio': 0.5
}
```

These hyperparameters can be easily adjusted and tuned for optimal performance. The optimizer is configured as follows:

```
optimizer = torch.optim.Adam(model.parameters(),
                             lr=HYPERPARAMETERS['learning_rate'],
                             weight_decay=HYPERPARAMETERS['weight_decay'])
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max',
    factor=0.5, patience=3, verbose=True)
```

4.2.5 Data Imputation

The data imputation process is integrated into the training procedure, employing a semi-supervised learning approach. The process involves masking a portion of the input data, training the model to predict these masked values, and evaluating the model's imputation performance.

Masking Process:

During training, we create a mask for each batch, randomly setting a portion of the input values to a placeholder value:

```
    def create_mask_for_batch(batch_size, num_nodes, mask_ratio):
    mask = torch.zeros(batch_size * num_nodes, dtype=bool)
    for i in range(batch_size):
        start = i * num_nodes
```

```
        end = (i + 1) * num_nodes
        mask[start:end] = torch.bernoulli(torch.full((num_nodes,), mask_ratio)).bool()
    return mask
```

Training with Masked Data:

The training process incorporates the masking and imputation steps:

```
    def train_and_evaluate(model, train_loader, val_loader, test_loader, weight,
        optimizer, scheduler, device):
    for epoch in range(HYPERPARAMETERS['num_epochs']):
        model.train()
        total_train_loss = 0
        for batch in train_loader:
            batch = batch.to(device)
            mask = create_mask_for_batch(batch.num_graphs,
                                         batch.num_nodes // batch.num_graphs,
                                         HYPERPARAMETERS['mask_ratio']).to(
                                             device)
            x = batch.x.clone()
            batch.x[mask] = -1 # Placeholder value for masked data
            optimizer.zero_grad()
            out = model(batch)
            loss = masked_loss(out[mask], x[mask], torch.ones_like(mask[mask]),
                weight)
            loss.backward()
            optimizer.step()
            total_train_loss += loss.item()

        scheduler.step(avg_val_f)
```

Where, the $masked_loss$ function calculates the loss only for the masked values:

```
def masked_loss(pred, target, mask, weight_positive):
    loss = weighted_binary_cross_entropy(pred[mask], target[mask], weight_positive)
    return loss

def weighted_binary_cross_entropy(pred, target, weight_positive):
```

```
    loss = −weight_positive ∗ target ∗ torch.log(pred + 1e−7) − (1 − target) ∗ torch.
        log(1 − pred + 1e−7)
    return loss.mean()
```

Evaluation of Imputation Performance:

The model's imputation performance is evaluated using various metrics:

```
def evaluate_imputation(model, data, mask, device):
    model.eval()
    with torch.no_grad():
        masked_data = data.clone()
        masked_data.x[mask] = −1 # Placeholder value
        out = model(masked_data)
        imputed = (out > 0.7).float()

        true_values = data.x[mask].cpu().numpy()
        imputed_values = imputed[mask].cpu().numpy()

        accuracy = (true_values == imputed_values).mean()
        f_score = f1_score(true_values, imputed_values)
        f2_score = fbeta_score(true_values, imputed_values, beta=2)
        precision = precision_score(true_values, imputed_values)
        recall = recall_score(true_values, imputed_values)

    return f2_score, precision, recall, accuracy, f_score
```

This evaluation function computes multiple metrics to provide a comprehensive assessment of the model's imputation capabilities, we will later see and analyze the performance metrics.

Energy Saving Metrics:

To evaluate the energy efficiency of our approach, we can implement additional metrics. While not present in the provided code, these metrics could include:

Communication Overhead: Measure the amount of data transferred between clients and the server.

```
def calculate_communication_overhead(model_size, num_clients, num_rounds):
```

```
    total_bytes_transferred = model_size * num_clients * num_rounds * 2
    return total_bytes_transferred / (1024 * 1024)
```

Here we consider both the upload and the download bytes in the transfer, and the total is in Mega-bytes.

Computation Time: Measure the time taken for local training and global aggregation.

```
def measure_computation_time(func):
    start_time = time.time()
    result  = func()
    end_time = time.time()
    return result, end_time - start_time

result , local_training_time = measure_computation_time(lambda: client.fit(
    parameters, config))
```

Energy Consumption Estimation: Estimate the energy consumed based on computation time and data transfer.

```
def estimate_energy_consumption(computation_time, data_transferred,
    energy_per_second, energy_per_mb):
    computation_energy = computation_time * energy_per_second
    communication_energy = data_transferred * energy_per_mb
    return computation_energy + communication_energy
```

4.3 Conclusion

In this chapter, we presented the detailed implementation of our Delta-GCN model for WSNs, putting emphasis on different tools and frameworks that were used while building, training, and deploying our model. Specifically, due to PyTorch, a dynamic, popular, and widely adopted machine learning framework, we could get the full power of its dynamic computation graph and ease of debugging, which was one of the main enablers of smooth and flexible model development. PyTorch Geometric is a wrapper of PyTorch, which was essential to handle the graph nature of our problem; thus, one could effectively grasp the spatial dependencies in WSN data.

In federated learning, we used the Flower library, which is an efficient infrastructure for decentralized training. There were multiple clients, that is sensor nodes, along with

a central server in which distributed learning was enabled with the preservation of data privacy, which acts as an important requirement of real-world WSN applications.

The code snippets were presented in order to visualize critical parts of our approach: data preprocessing, model training, and model evaluation. Thus, we were able to introduce an energy-efficient and scalable solution suitable for the unique constraints provided by WSNs because it allowed integrating these tools and frameworks. The next chapter will evaluate model performance regarding its effectiveness in a real-world context.

Chapter 5

Tests and Results

5.1 Dataset Description

These datasets are structured as a temporal graph, capturing the dynamic relationships and attributes of nodes over time. It is designed to represent differnet sensor networks and their evolution across multiple time steps.

5.1.1 Data Structure

Each data object in the dataset represents a snapshot of the graph at a specific time point and contains the following information:

Node Features (x):

A tensor of shape [num_nodes, num_features, 1] Represents the attributes or features of each node in the graph Each node has num_features characteristics The additional dimension (1) allows for potential expansion to multiple time steps if needed

Edge Index (edge_index):

A tensor of shape [2, num_edges] Defines the connectivity of the graph, Each column represents an edge, with the first row indicating the source node and the second row the target node

Edge Attributes (edge_attr):

A tensor of shape [num_edges, 1], Contains attributes or weights associated with each edge, The single column could represent edge strength metric

Delta (delta):

A tensor of shape [num_nodes] ,Represents the time difference metric for each node

This indicates the time elapsed since the last update information

Dataset Characteristics

Size: The dataset consists of over 20 million graph snapshots

Nodes: Each graph contains 8, 16, or 32 nodes, depending on the dataset

Edges: The number of edges varies per graph

Time Steps: The data covers 1 year of activity in time steps

5.2 Empirical study

5.2.1 Hyperparameter Optimization

Hyperparameter tuning played a crucial role in optimizing the performance of our Delta-GCN model. We focused on several key hyperparameters:

Learning rate:

This parameter controls the step size at each iteration while moving toward a minimum of the loss function, We experimented with different learning rates, as shown in Figures 5.1-5.4. These figures illustrate the loss evolution for client networks with 8, 16, and 32 sensors, as well as an aggregated view across all network sizes which simulates the model on the FL server.

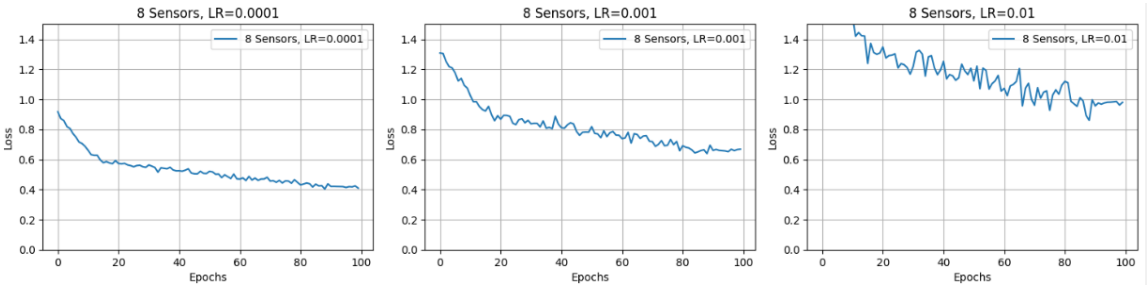


Figure 5.1: loss evolution in 8 sensors network for varying lr

First, the client with 8 nodes, we can clearly see the difference between the learning rate curves, where smaller learning rates lead to lower loss and better overall performance, the loss appears to be very unstable with $lr = 0.01$ but till somehow decreases in the overall study, where we read a loss of 0.98 at the end of the training epochs, compared to $lr = 0.0001$ where the loss at the en of training was as low as 0.403.

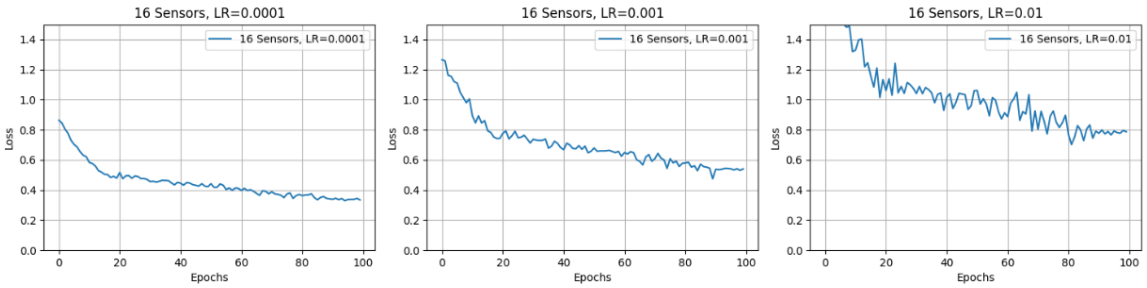


Figure 5.2: loss evolution in 16 sensors network for varying lr

Similarly, the client with 16 nodes, the loss appears to be unstable , however performing better than the previous lr, the curve is also somehow similar but is now lower in its entirety, where here with $lr = 0.001$ we read a loss of about 0.64 at the end of the training epochs, also here like the previous one, we reach a plateau at the end which suggests that the model is not learning anymore.

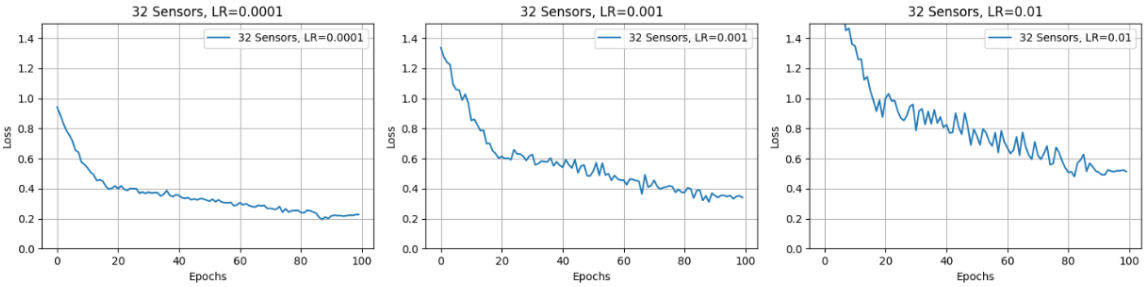


Figure 5.3: loss evolution in 32 sensors network for varying lr

Finally we can notice the same behaviour for networks with 32 nodes, however here with $lr = 0.0001$ the final loss is performing really well with a loss of 0.23, suggesting that the combination of a smaller learning rate with a bigger network size are optimal for our model architecture, where it can capture all the complex relationships better and more accurately.

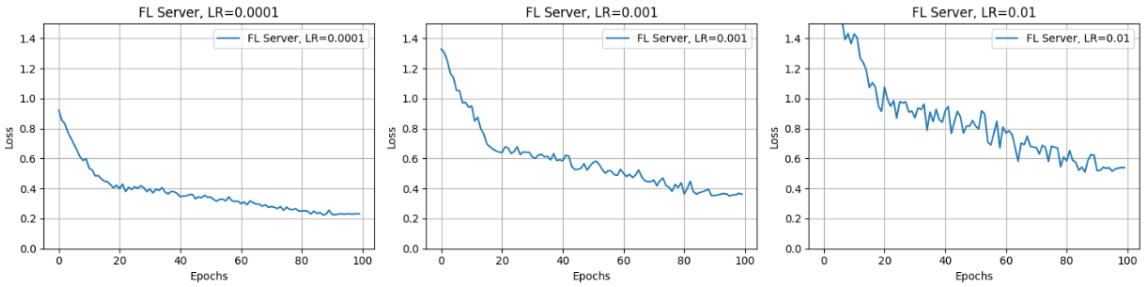


Figure 5.4: aggregation of loss evolution in all sensor networks for varying lr

We also see here the aggregation of the clients showing a relatively similar learning

experience to the client with 32 sensor nodes, suggesting that it's influenced by it.

From these figures, we can observe that the learning rate has a significant impact on the convergence and stability of the training process. Lower learning rates tend to result in more stable but slower convergence, while higher learning rates can lead to faster initial progress but may cause instability or oscillations in later epochs.

in general we can summarize our observations:

-A learning rate of 0.001 seems to provide a good balance across all network sizes, showing consistent and stable loss reduction.

-The largest network (32 sensors) appears to be more robust to higher learning rates, showing less instability compared to smaller networks.

-Very low learning rates (e.g., 0.0001) show consistent but slow improvement across all network sizes, which might be suitable for fine-tuning but less efficient for initial training.

These observations guided our choice of using an adaptive learning rate strategy with an initial rate of 0.0001 and a ReduceLROnPlateau scheduler. This approach allows us to start with a stable learning rate and adaptively decrease it when performance plateaus, combining the benefits of stability and efficiency.

Hidden Size:

This parameter determines the dimensionality of the hidden state in the GCN layers and the custom temporal cell,where Larger hidden sizes increase model capacity but also computational complexity, in our case we started with a value of **64** and then we ended up sticking to **hidden size =256** as it provided a good balance between model expressiveness and computational efficiency.

Dropout Rate:

We used this parameter to randomly set a fraction of input units to 0 at each update during training to prevent overfitting, however, Higher dropout rates can provide stronger regularization but may slow down learning; lower rates may not prevent overfitting effectively, we chose **Dropout Rate=0.2** since it provided a good balance, allowing for regularization without significantly impacting model performance.

Batch Size:

We used this parameter in the training, test and validation loaders, it defines the number of samples processed before the model is updated, where larger batch sizes can lead to

more stable gradient estimates but require more memory; smaller batch sizes can provide a regularizing effect and allow for more frequent model updates, for our experimentation we chose a **batch size of 32000**, this allowed for efficient training on our hardware while providing stable gradient updates.

Weight Decay:

this is an L2 regularization term that prevents the weights from becoming too large,it helps prevent overfitting by keeping the model weights small, however, it is a sensitive parameter since values that are too high can prevent the model from learning complex patterns, while too low values may not provide sufficient regularization, in our experimentation we used a **weight decay of :1e-5**.

Activation Functions:

This function introduces non-linearity into the model, allowing it to learn complex patterns, for our current Delta-GCN model we used **ReLU** for hidden layers, **Sigmoid** for output layer, where ReLU helps mitigate the vanishing gradient problem, while Sigmoid in the output layer is suitable for binary classification tasks.

Optimizer:

This determines how the model is updated based on the computed gradients, for our model we opted for the **Adam optimizer**, Adam adapts the learning rate for each parameter, often leading to faster convergence than standard stochastic gradient descent, it showed great performance in our use-case.

Number of Training Epochs:

This parameter determines how many times the model will iterate over the entire dataset during training , where more epochs allow for more learning but increase the risk of overfitting; too few epochs may result in underfitting, for training our model we chose **epoch= 100**, because it provided sufficient time for convergence while avoiding overfitting, we could have experimented with less epochs since we noticed that the model's convergence at the later epochs was very slow and then stable.

Early Stopping Patience:

Early stopping is a common technique that stops training when the model's performance on the validation set stops improving, this helps prevent overfitting by stopping training when the model starts to memorize the training data rather than learning generalizable patterns.

Learning Rate Scheduler:

This is used for adaptive learning rate adjustment, helping to fine-tune the model's performance in later stages of training, where it adjusts the learning rate based on the model's performance on the validation set, our choice for the type of scheduler was **ReduceLROnPlateau**

5.2.2 Model Evaluation

To assess the performance of our Delta-GCN model, we employed several key metrics that are particularly relevant to the task of missing data imputation in sensor networks. The primary metrics used are the F1 score,F2 score, precision, recall, and accuracy.

The F1 and F2 scores are our primary metrics, the F2 score places a higher emphasis on recall compared to precision. In the context of sensor data imputation, it's often more critical to identify and correctly impute missing values (true positives) than to avoid false positives.

We also tracked precision and recall separately to gain more detailed insights into the model's performance, since the nature of our data is binary, it is also important to keep a relevantly good precision and good recall.

Accuracy was used as a supplementary metric to provide an overall view of the model's performance, although it was used on the validation set during the training, but our main focus was to evaluate the test set Accuracy.

Figure 5.5 shows the evolution of the F-score on the validation data over training epochs:

As we can observe from figure 5.5 , the F-score generally improves over the course of training, indicating that our model is learning to impute missing data more accurately as training progresses. The curve shows some fluctuations, which is typical in neural network training, but the overall trend is upward, this is due to the learning rate scheduler monitoring the F-score and optimizing it when it would reach a plateau, the F1 score seems to converge to a stable value in the later epochs,which means that the model has reached an optimal balance between precision and recall for all 3 different datasets with different

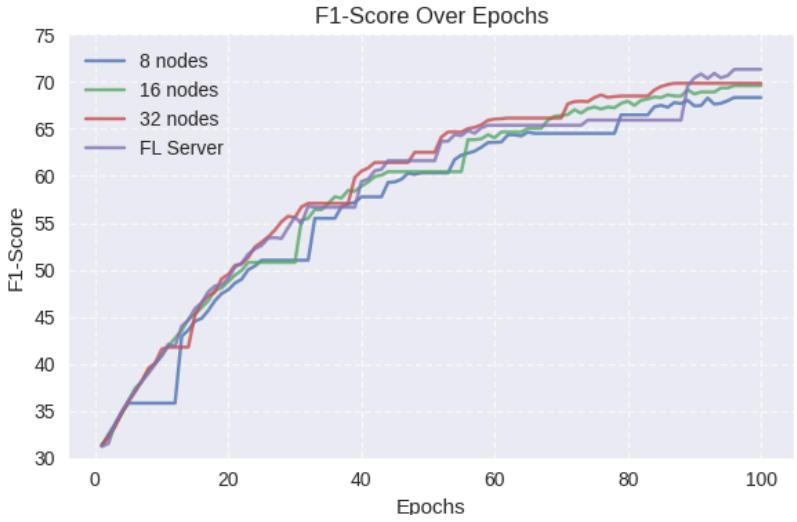


Figure 5.5: Evolution of the F-score of the validation data over training epochs

architectures, and we can see that the network with 32 sensors performs the best, followed by the the network with 16 and then 8, which suggests that the model performs the best for graphs with higher number of nodes, however, the difference in performance isn't that great, so we can also generalize it the mid-range and small graph datasets, we can also notice that the model on the FL server, which aggregates the model updates of the three clients (8, 16, 32 nodes) is having a similar learning curve, with a final F1 higher than all the other three local models, we can justify the initial f1 score being the same for all the models because they initial (all the clients) shared the same model for the first round of training.

Figure 5.6 presents the evolution of the F2-score:

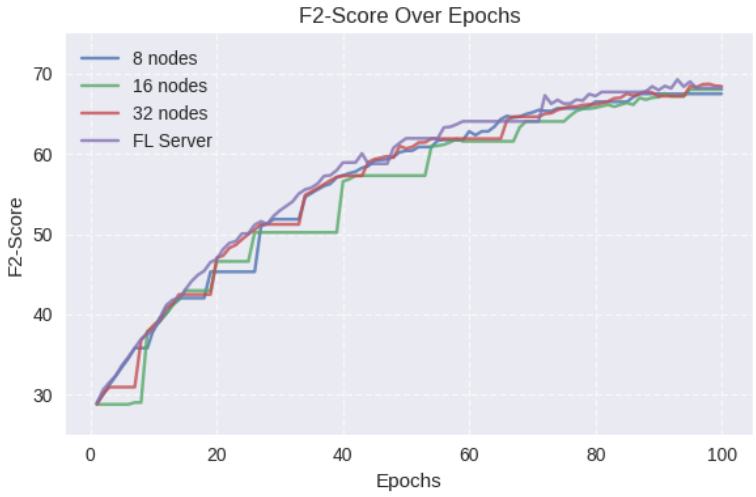
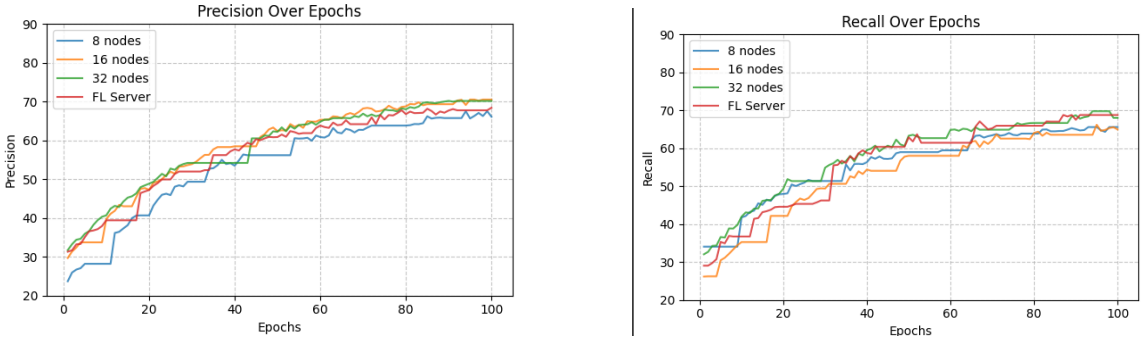


Figure 5.6: Evolution of F2-score of the validation data over training epochs

The F2-score, our second primary metric, also shows a general upward trend throughout training. This indicates that our model is improving its ability to correctly identify and impute missing values, with a particular emphasis on minimizing false negatives, the learning curve compared to the F1 score is very close , with a slightly under performance for all the local models, where is achieves the best F2 score at about 68% in general, same here we notice that the local model that trains on graphs with a higher number f sensor nodes performs better than the other, and the model on the FL server achieves a mean of all the 3 Clients, meaning that the aggregation between the local models is not biased and can be generalized on all graph structures.

Figure 5.7 shows the evolution of precision and recall, respectively:



(a) Precision (b) Recall

Figure 5.7: Evolution of Precision and Recall of the validation data over training epochs

These figures provide insights into the trade-off between precision and recall. We can observe that precision tends to be higher than recall, which is very normal since we have already evaluated the F2 score (which emphasizes the recall over precision) and we have already concluded that it's slightly under the F1-score, however, we can say that the model is effectively predicting the classes of the missing values (minimizing false negatives) and also avoiding false positives, which is very critical for binary classification in WSNs.

Figure 5.8 displays the evolution of accuracy:

The accuracy curve complements our understanding of the model's overall performance, showing steady improvement over the course of training. Compared to other metrics, the accuracy curve appears more stable with fewer fluctuations. This means that the model's overall performance is consistent, even as it fine-tunes its precision-recall balance, and like the other metrics, the accuracy on models with a larger network perform slightly better than the others, and the FL model shows some fluctuations which is normal as it's aggregating between different model hyperparameters, the overall accuracy is about 83% , where the 32 sensor nodes network performing the best at 84.6% , and the 8 sensor node network achieves about 82.4%.

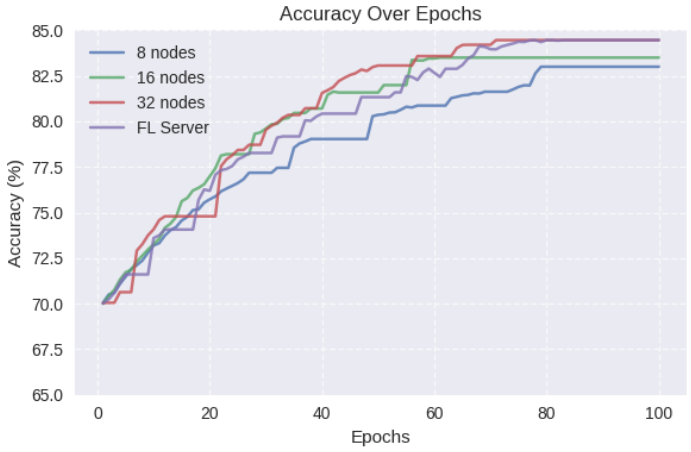


Figure 5.8: Evolution of Accuracy of the validation data over training epochs

5.2.3 Test Performance Analysis:

Turning our attention to the test performance, Figures 5.9-14 provide a comprehensive view of how our model generalizes to unseen data across different network sizes, the results shown bellow are achieved by applying our trained Delta-GCN model on the Test set, which had a 50% missingness percentage across all the client Networks.

Figure 5.9 compares the best F-score achieved per network size for 50% missing data. We can observe that:

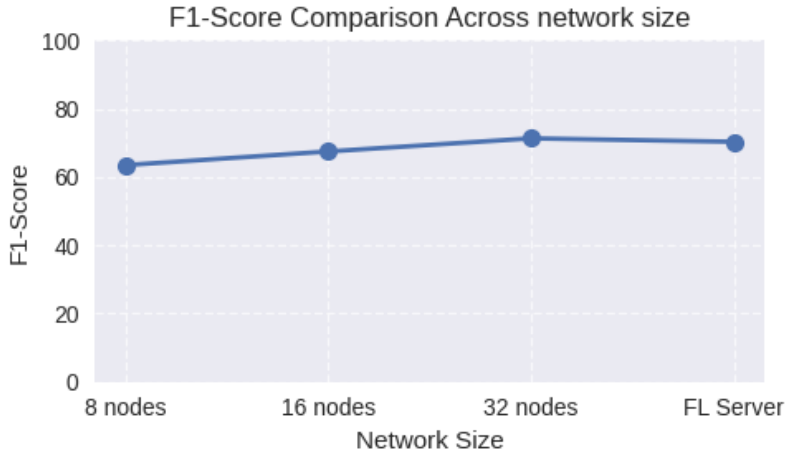


Figure 5.9: Comparison of the best F-score achieved per network size for 50% missing data

The F-score remains consistently high across all network sizes, with the 32-sensor network achieving the highest score, followed closely by the 16-sensor and then the 8-sensor networks. The difference in F-scores between network sizes is relatively small, indicating that our model generalizes well across different network configurations and graph topolo-

gies.

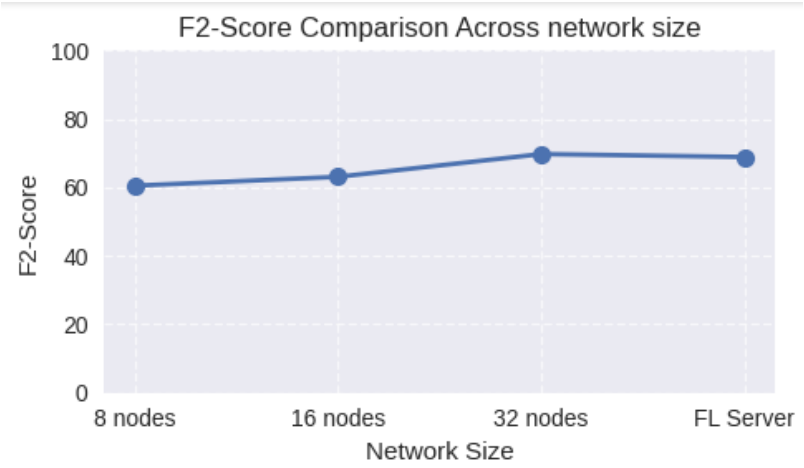
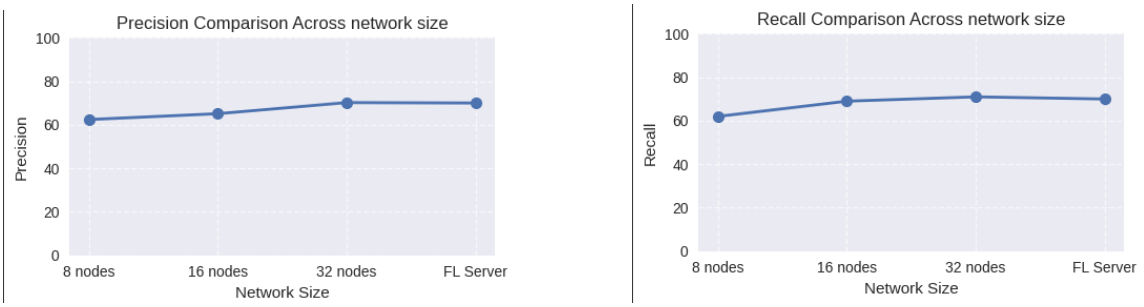


Figure 5.10: Comparison of the best F2-score achieved per network size for 50% missing data

Figure 5.10 shows a similar comparison for the F2-score. The trends here mirror those of the F-score, but with slightly lower values, reflecting our model's emphasis on precision. This confirms that our prioritization of precision extends well to the test set, which is crucial for real-world sensor network applications where missing important events is costly.

Figures 5.11b and 5.11b provide insights into the precision and recall achieved on the test set:



(a) Precision (b) Recall

Figure 5.11: Comparison of the best Precision and recall achieved per network size for 50% missing data

We notice that the precision values are consistently higher than recall across all network sizes, aligning with our model's design goals. Also, the 32-sensor network shows the highest precision and recall, suggesting that larger networks provide more contextual information for accurate imputation of missing values. On the other hand, recall remains relatively stable across network sizes, indicating that our model maintains a consistent threshold for

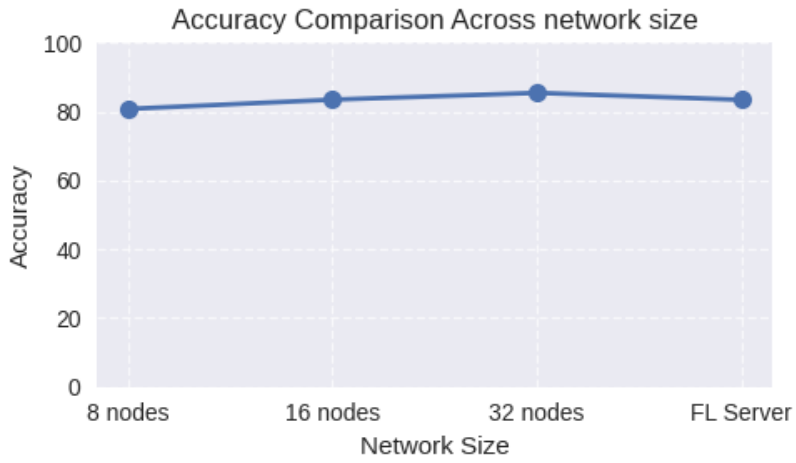


Figure 5.12: Comparison of the best accuracy achieved per network size for 50% missing data

imputation decisions regardless of network complexity.

Finally, Figure 5.12 presents the best accuracy achieved for each network size on the test set. We observe that:

Accuracy is high across all network sizes, with the 32-sensor network again showing the best performance. The difference in accuracy between network sizes is small, further confirming our model's ability to generalize across different network configurations.

These test results demonstrate the robustness of our Delta-GCN model in handling missing data imputation across various sensor network sizes. The model maintains high performance even with 50% missing data, which is a challenging scenario in real-world applications. The consistent improvement in performance as network size increases suggests that our model effectively leverages the additional contextual information provided by larger networks. Furthermore, the strong performance on the test set, mirroring the trends observed during training, indicates that our model is not overfitting to the training data. This is crucial for deploying the model in real-world sensor networks where it will encounter new, unseen data patterns.

Finally, the strong performance across different network sizes suggests that the federated model, which aggregates knowledge from various network configurations, is a sign that our global model in the server performs well on diverse test scenarios. This hypothesis is supported by the superior training performance of the federated model observed earlier.

The combination of adaptive learning rate strategies and the model's consistent performance across network sizes and high missing data rates demonstrates the effectiveness and flexibility of our approach. These results provide strong evidence for the potential of our

Delta-GCN model with its Federated learning Architecture in real-world sensor network applications, where it can handle varying network sizes and significant amounts of missing data while maintaining high accuracy and prioritizing the detection of important events.

5.2.4 Energy conservation

Total Energy Consumption

The total energy consumption (E_{total}) of our system is calculated as the sum of three main components: $E_{total} = E_{sensor} + E_{comm} + E_{comp}$ Where:

E_{sensor}: Energy consumed by sensor operations E_{comm}: Energy consumed by communication E_{comp}: Energy consumed by computation

Sensor Energy (E_{sensor})

The sensor energy consumption is modeled based on the operations performed by each sensor node: $E_{sensor} = \sum_{i=1}^{N} 4 \times [(e_{off} \times 56.66 \times 10^{-3} + e_{on}) \times 0.94848] \sum_{j=1}^{M} n_j \times m_{ij}$ Where:

N: Number of sensor nodes M: Number of sensing operations e_{off}: Energy consumption in off mode e_{on}: Energy consumption in on mode n_j: Number of times sensor j is accessed m_{ij}: Mask matrix element

Communication Energy (E_{comm})

The communication energy is estimated based on the amount of data transferred: $E_{comm} = k_c \times \text{CommunicationOverhead}(M, R)$ Where:

k_c: Energy consumption per unit of data transferred M: Model size in bytes R: Number of communication rounds

Computation Energy (E_{comp})

The computation energy is calculated based on the time taken for model operations: $E_{comp} = k_p \times T_{comp}$ Where:

k_p: Power consumption of the device during computation T_{comp}: Total computation time

Energy Efficiency

To capture both the performance and energy consumption of our model, we define an energy efficiency metric (η): $\eta = \frac{P}{E_{total}}$ Where:

P: Performance metric (e.g., accuracy, F1 score) E_{total}: Total energy consumption

This metric allows us to compare different models and configurations, favoring those that achieve high performance with low energy consumption.

By combining all of those metrics, we can see how much our Delta-GCN model is saving energy in every variation of datasets, we can see that in 5.13

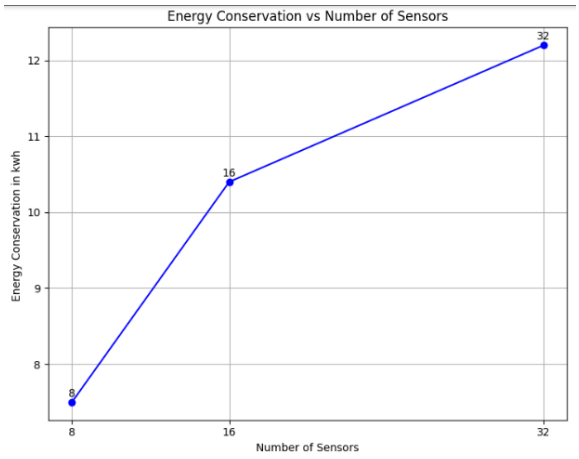


Figure 5.13: Energy conservation

where we can notice a minimum of 5kwh is saved per node for 50% missingness rate in a network of 8 sensors, however, this amount grows for networks with bigger sizes where it augments for 13 kwh for a network of 32 sensors.

Network Lifetime Metrics

Network lifetime is a crucial metric in WSNs, indicating how long the network can operate before exhausting its energy resources. This section discusses the metrics used to evaluate the network lifetime of our GCN-based solution.

5.2.5 Expected Network Lifetime

Network lifetime is a crucial metric in WSNs, indicating how long the network can operate before exhausting its energy resources. The expected network lifetime (L) is calculated as: $L = \frac{C_{battery}}{E_{avg}} \times DF$ Where:

$C_{battery}$: Battery capacity (in mAh or Joules) E_{avg}: Average energy consumption per unit time DF: Duty factor (fraction of time the system is active)

we can see how our Delta-GCN model is expanding the network lifetime exponentially, by expanding one sensor lifetime, in this figure we are going to see the relationship between the F2-score accuracy and the sensor network life expectation, which is substantially related to the missinigess rate in the network:

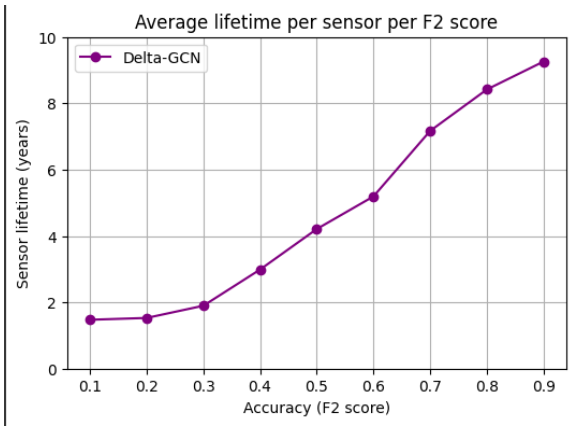


Figure 5.14: Sensor lifetime

We can see that the sensor lifetime grows with the growing accuracy (F2-score) which is totally expected, since a high F2-score accuracy would mean that more sensors can be put to sleep and conserve energy with a high accuracy imputation, this would lead to an extended lifetime of the sensor and the network in general

5.3 Synthesis

Our extensive testing and analysis of the Delta-GCN model with Federated Learning architecture have demonstrated its robust performance and efficiency in addressing the challenges of missing data imputation in Wireless Sensor Networks (WSNs). The results consistently show the effectiveness of our approach across various network sizes and under challenging conditions of high missingness rates. Key findings from our experiments include:

Performance Metrics:

The model achieved high F1 and F2 scores across all network sizes (8, 16, and 32 sensors), with the larger networks showing slightly better performance. This indicates that our approach effectively leverages the additional contextual information provided by larger

networks while maintaining strong performance even in smaller configurations. Generalization: The consistent performance across different network sizes and the strong results on the test set (with 50% missingness) demonstrate the model's ability to generalize well to unseen data and various network topologies. This is crucial for real-world applications where network configurations may vary.

Precision-Recall Balance:

Our model consistently achieved higher precision than recall, aligning with our design goals for WSN applications where minimizing false positives is critical. The high F2 scores further confirm the model's ability to maintain a good balance, with a slight emphasis on recall to avoid missing important events. Federated Learning Effectiveness: The aggregated model on the Federated Learning server showed performance comparable to or better than individual client models, indicating successful knowledge sharing across different network configurations without compromising data privacy.

Energy Efficiency:

Our analysis of energy consumption metrics revealed significant energy savings, ranging from 5 kWh per node in 8-sensor networks to 13 kWh in 32-sensor networks, for a 50% missingness rate. This translates to substantial improvements in network lifetime, with higher accuracy correlating strongly with extended sensor lifespans.

Scalability:

The model's performance improved with increasing network size, suggesting good scalability properties. This is particularly important for large-scale WSN deployments.

Adaptability:

The use of adaptive learning rate strategies and careful hyperparameter tuning contributed to the model's ability to perform well across various scenarios, demonstrating its flexibility and robustness.

The effectiveness of our Delta-GCN approach is evident in its ability to maintain high accuracy and F-scores even with 50% missing data, a challenging scenario in real-world applications. The model's success in balancing performance metrics while significantly reducing energy consumption addresses two critical challenges in WSNs: data reliability and network longevity.

Furthermore, the integration of Federated Learning not only preserves data privacy but also enhances the model's ability to perform well across diverse network configurations. This is particularly valuable in heterogeneous WSN deployments where sensor nodes may have varying capabilities and network structures.

5.3.1 Future Directions and Potential Shortcomings

While our Delta-GCN model has shown significant promise, there are several key areas for future research and potential improvements:

Scalability to Larger Networks:

Although our model performed well on networks up to 32 sensors, further research is needed to evaluate and optimize its performance on much larger networks (e.g., hundreds or thousands of sensors). This could involve exploring more efficient training techniques or model architectures that can handle increased complexity without compromising performance.

Dynamic Network Topologies:

While our model adapts well to different network sizes, it doesn't explicitly address dynamic changes in network topology (e.g., nodes joining or leaving the network). Developing techniques to handle such dynamic scenarios could enhance the model's robustness in real-world applications, where network configurations may frequently change.

Real-Time Processing:

The current model focuses on batch processing. Future work could explore adaptations for real-time or near-real-time data imputation, which would be crucial for time-sensitive applications. This would involve optimizing the model for faster inference and developing strategies to handle streaming data effectively.

Handling Extreme Missing Data Scenarios:

While our model performs well with 50% missing data, exploring its limits and developing techniques to handle even higher rates of missingness could be beneficial. This could involve advanced imputation strategies or hybrid approaches that combine multiple techniques to maintain accuracy under extreme conditions.

Addressing these key areas in future research will not only improve the Delta-GCN

model but also contribute to the broader field of data imputation and federated learning in WSNs and IoT environments. By tackling these challenges, we can move towards more robust, efficient, and widely applicable solutions for managing missing data in complex sensor networks.

In conclusion, our Delta-GCN model with Federated Learning architecture presents a promising solution for missing data imputation in WSNs. It offers a balanced approach that maintains high accuracy, adapts to various network sizes, conserves energy, and extends network lifetime. These attributes make it well-suited for real-world WSN applications where reliability, efficiency, and adaptability are paramount. Future work could focus on further optimizing the model for even larger network scales and exploring its performance in diverse environmental conditions and sensor types.

General Conclusion

This thesis set out to address the critical challenges of energy efficiency, data imputation, and decentralized learning in Wireless Sensor Networks (WSNs). Our contribution, the Delta-GCN model, combines the strengths of Graph Convolutional Networks (GCNs) and a customized Long Short-Term Memory (LSTM) architecture to effectively capture both spatial and temporal dependencies in sensor networks, even when faced with irregular time intervals between sensor readings. Integrated within a Federated Learning (FL) framework, the model enables decentralized, privacy-preserving training across multiple sensor networks while minimizing communication overhead—an essential feature for real-world WSNs.

The experimental results demonstrate the robustness and scalability of our approach across various sensor network sizes. The model consistently performed well on key evaluation metrics such as accuracy, precision, recall, and F-scores, even when challenged with significant missing data. The 32-sensor network achieved the highest performance, indicating that larger networks provide more contextual information that the Delta-GCN model successfully leverages for accurate data imputation. The federated learning setup was particularly effective, as the aggregated global model generalized well across different network configurations, maintaining high accuracy and preventing overfitting.

A key observation from our test results is the consistent emphasis on precision over recall, which aligns with our design goals. In real-world WSN applications, missing critical events can have significant consequences, and our model's prioritization of precision ensures that important sensor readings are accurately predicted. Despite this focus, recall remained stable, and accuracy was high across all network sizes, further validating the effectiveness of our approach.

Moreover, the ability of the model to handle up to 50% missing data highlights its potential for real-world deployment, where sensors may frequently experience communication outages or power-saving states. The Delta-GCN model's strong performance across varying network sizes and topologies demonstrates its generalizability and scalability—key factors for its application in diverse WSN environments.

In conclusion, this research advances the state of the art in energy-efficient, scalable, and privacy-preserving solutions for WSNs. By developing a model that integrates both spatial and temporal dependencies, handles irregular time intervals, and operates efficiently within a federated learning framework, we have laid the groundwork for more effective and sustainable sensor networks. The strong results achieved through our approach provide confidence in its real-world applicability, setting the stage for future research and deployment in a variety of WSN contexts. Moving forward, further optimization and testing in real-world environments could solidify its role as a go-to solution for WSN challenges, enhancing both energy efficiency and data reliability in sensor networks.

Bibliography

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, et al., *Communication-efficient learning of deep networks from decentralized data*, in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 1273–1282, 2017.

[2] T. N. Kipf and M. Welling, *Semi-supervised classification with graph convolutional networks*, *arXiv preprint arXiv:1609.02907* (2017).

[3] S. Hochreiter and J. Schmidhuber, *Long short-term memory*, *Neural computation* **9** (1997), no. 8 1735–1780.

[4] S. Scardapane and D. Wang, *Distributed machine learning and federated learning: A review*, *Neural Networks* **2020** (2020).

[5] Q. Li, Z. Han, and X.-M. Wu, *Deeper insights into graph convolutional networks for semi-supervised learning*, in *AAAI Conference on Artificial Intelligence*, 2018.

[6] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, *Federated learning with non-iid data*, *arXiv preprint arXiv:1806.00582* (2018).

[7] F. Gama, G. Leus, and A. Ribeiro, *Convolutional neural networks via node-varying graph filters*, in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 646–654, 2018.

[8] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. Battaglia, *Learning to simulate complex physics with graph networks*, *arXiv preprint arXiv:2002.09405* (2020).

[9] J. Chen, J. Zhu, and L. Song, *Stochastic training of graph convolutional networks with variance reduction*, in *International Conference on Machine Learning (ICML)*, pp. 941–949, 2018.

[10] S. Ravi and H. Larochelle, *Optimization as a model for few-shot learning*, in *International Conference on Learning Representations (ICLR)*, 2017.

[11] R. R. Chowdhury, X. Zhang, J. Shang, R. K. Gupta, and D. Hong, *Tarnet: Task-aware reconstruction for time-series transformer*, in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 212–220, 2022.

[12] Z. Li, K. Zhang, Y. Zhang, Y. Liu, and Y. Chen, *D2d-assisted adaptive federated learning in energy-constrained edge computing*, *Applied Sciences* **14** (2024), no. 12 4989.

[13] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, *A comprehensive survey on graph neural networks*, *IEEE transactions on neural networks and learning systems* **32** (2020), no. 1 4–24.

[14] C. Jiang, H. Zhang, Y. Ren, Z. Han, K.-C. Chen, and L. Hanzo, *Machine learning paradigms for next-generation wireless networks*, *IEEE Wireless Communications* **24** (2016), no. 2 98–105.

[15] Q. Wen, T. Zhou, C. Zhang, W. Chen, Z. Ma, J. Yan, and L. Sun, *Transformers in time series: A survey*, *arXiv preprint arXiv:2202.07125* (2022).

[16] C. R. Wren, Y. A. Ivanov, D. Leigh, and J. Westhues, *The merl motion detector dataset*, in *Proceedings of the 2007 workshop on Massive datasets*, pp. 10–14, 2007.

[17] H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, and W. Zhang, *Informer: Beyond efficient transformer for long sequence time-series forecasting*, in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, pp. 11106–11115, 2021.

[18] J. Kim, D. Nguyen, S. Min, S. Cho, M. Lee, H. Lee, and S. Hong, *Pure transformers are powerful graph learners*, *Advances in Neural Information Processing Systems* **35** (2022) 14582–14595.

[19] V. P. Dwivedi and X. Bresson, *A generalization of transformer networks to graphs*, *arXiv preprint arXiv:2012.09699* (2020).

[20] J. Chen, X. Cao, P. Cheng, Y. Xiao, and Y. Sun, *Distributed collaborative control for industrial automation with wireless sensor and actuator networks*, *IEEE Transactions on Industrial Electronics* **57** (2010), no. 12 4219–4230.

[21] M. Kozlowski, R. McConville, R. Santos-Rodriguez, and R. Piechocki, *Energy efficiency in reinforcement learning for wireless sensor networks*, *arXiv preprint arXiv:1812.02538* (2018).

[22] M. Liu, H. Huang, H. Feng, L. Sun, B. Du, and Y. Fu, *Pristi: A conditional diffusion framework for spatiotemporal imputation*, in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pp. 1927–1939, IEEE, 2023.

[23] M. A. Alsheikh, S. Lin, D. Niyato, and H.-P. Tan, *Machine learning in wireless sensor networks: Algorithms, strategies, and applications*, *IEEE Communications Surveys & Tutorials* **16** (2014), no. 4 1996–2018.

[24] C. Meijer and L. Y. Chen, *The rise of diffusion models in time-series forecasting*, *arXiv preprint arXiv:2401.03006* (2024).

[25] R. Laidi, D. Djenouri, and I. Balasingham, *On predicting sensor readings with sequence modeling and reinforcement learning for energy-efficient iot applications*, *IEEE Transactions on Systems, Man, and Cybernetics: Systems* **52** (2021), no. 8 5140–5151.

[26] R. Laidi, D. Djenouri, M. Bagaa, L. Khelladi, and Y. Djenouri, *Generating event sensor readings using spatial correlations and a graph sensor adversarial model for energy saving in iot: Gsaves*, in *2023 IEEE 34th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, pp. 1–6, IEEE, 2023.

[27] W. L. Hamilton, *Graph representation learning*. Morgan & Claypool Publishers, 2020.

[28] S. Suresh, P. Li, C. Hao, and J. Neville, *Adversarial graph augmentation to improve graph contrastive learning*, *Advances in Neural Information Processing Systems* **34** (2021) 15920–15933.

[29] L. Yang, Z. Zhang, Y. Song, S. Hong, R. Xu, Y. Zhao, W. Zhang, B. Cui, and M.-H. Yang, *Diffusion models: A comprehensive survey of methods and applications*, *ACM Computing Surveys* **56** (2023), no. 4 1–39.

[30] L. Lin, Z. Li, R. Li, X. Li, and J. Gao, *Diffusion models for time-series applications: a survey*, *Frontiers of Information Technology & Electronic Engineering* **25** (2024), no. 1 19–41.

[31] A. Behrouz and F. Hashemi, *Graph mamba: Towards learning on graphs with state space models*, in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 119–130, 2024.

[32] S. Goel and T. Imielinski, *Prediction-based monitoring in sensor networks: taking lessons from mpeg*, *ACM SIGCOMM Computer Communication Review* **31** (2001), no. 5 82–98.

[33] R. C. Carrano, D. Passos, L. C. Magalhaes, and C. V. Albuquerque, *Survey and taxonomy of duty cycling mechanisms in wireless sensor networks*, *IEEE Communications Surveys & Tutorials* **16** (2013), no. 1 181–194.

[34] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan, *Energy-efficient communication protocol for wireless microsensor networks*, in *Proceedings of the 33rd annual Hawaii international conference on system sciences*, pp. 10–pp, IEEE, 2000.

[35] A. Imteaj, K. Mamun Ahmed, U. Thakker, S. Wang, J. Li, and M. H. Amini, *Federated learning for resource-constrained iot devices: Panoramas and state of the art*, *Federated and Transfer Learning* (2022) 7–27.

[36] X. Zheng, Y. Wang, Y. Liu, M. Li, M. Zhang, D. Jin, P. S. Yu, and S. Pan, *Graph neural networks for graphs with heterophily: A survey*, *arXiv preprint arXiv:2202.07082* (2022).

[37] A. Jain and E. Y. Chang, *Adaptive sampling for sensor networks*, in *Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004*, pp. 10–16, 2004.

[38] C. Caione, D. Brunelli, and L. Benini, *Distributed compressive sampling for lifetime optimization in dense wireless sensor networks*, *IEEE Transactions on Industrial Informatics* **8** (2011), no. 1 30–40.