POLITECNICO DI TORINO

Master's degree in electronic engineering

Master's degree thesis

Enhancing Indoor Human Localization with Echo State Networks and Temporal Convolutional Networks

Supervisor:

Professor Mihai Teodor Lazarescu

Candidate:

Sina Einavi Pour

October 2024

Abstract

Indoor human localization has gained significant importance due to its wide range of applications, including smart environments, health monitoring, and security systems. However, achieving accurate indoor positioning remains a challenging task due to complex environments, signal interference, and the demand for real-time processing. Traditional approaches such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) models are commonly used to process time-series data from multiple sensors. Yet, these models suffer from drawbacks like high resource consumption and vanishing/exploding gradient issues, especially in environments with real-time constraints. To address these challenges, this thesis presents an innovative solution using Echo State Networks (ESN) and Temporal Convolutional Networks (TCN) with capacitive sensors for indoor localization.

The motivation behind using ESNs lies in their computational efficiency, where only the output weights are trained while the random reservoir remains fixed. This drastically reduces the computational cost compared to traditional recurrent models, making ESNs well-suited for real-time localization in resource-constrained environments such as IoT systems. TCNs, on the other hand, offer superior performance in modeling long-term dependencies using dilated convolutions, which also enable parallel computation, addressing the gradient vanishing/exploding problems. This combination of ESNs and TCNs aims to balance both accuracy and smoothness for indoor localization tasks, using minimal computational resources.

The experimental setup was designed to mimic a realistic indoor environment, specifically an empty 3 m × 3 m room. Capacitive sensors were placed at chest height, with each sensor featuring a 16 cm × 16 cm sensing plate operating in load mode. These sensors capture the capacitance changes induced by human proximity, providing three readings per second (3 Hz). Then an indirect measurement of the capacitance has been performed by measuring the frequency of a 555 timer-based relaxation oscillator. The capacitive coupling between the human body and the sensors allows distance to be inferred from the measured capacitance. In addition, a reference system based on ultrasound anchors was employed to track the person's exact location with ±2 cm accuracy at 15 Hz, providing ground truth data for performance evaluation.

To ensure accurate localization predictions, the capacitive sensor data underwent several preprocessing steps:

1. The average of the sensor readings was translated to zero.
2. The data was filtered using a 50-second Median Filter (MF) to extract slow drift.
3. A Low-Pass Filter (LPF) with a pass-band edge of 0.1 Hz and a stop-band edge of 0.6 Hz was applied to reduce high-frequency noise.
4. The MF output was subtracted from the LPF output to clean the signal.
5. The resulting values were normalized to the range [0, 1], preparing the data for training neural networks.

Three architectures were evaluated in this study:

- ESN: Consisting of a random fixed reservoir of neurons, the ESN processes the sensor data and predicts coordinates. Its simplicity and low parameter count make it efficient for real-time applications. In this model the outputs of 4 capacitive sensors are fed into the reservoir and the ESN predicts the x, y coordinates.
- TCN: The TCN employs dilated convolutions to model long-term dependencies, providing a baseline for comparison in terms of accuracy and smoothness. A window of 15 samples from 4 sensors are fed into TCN and the x, y coordinates of the 8th sample is predicted by the network.
- Hybrid TCN-ESN: A novel architecture combining TCN's long-range temporal modeling capabilities with ESN's computational efficiency. This model achieves superior performance in both accuracy and smoothness but requires more resources than either model individually. In this model the TCN is followed by a 2 input ESN. The TCN receives the values from the 4 sensors (a window of length 15) and predicts the x, y coordinates of the 8th sample. Then these x, y are fed into the ESN to make the prediction more accurate.

The models have been trained and the hyperparameters have been tuned as follows:

- TCN: A Neural Architecture Search (NAS) was employed using AutoKeras to optimize the TCN architecture. The search focused on tuning the number of layers, filters, and dilations.

- ESN: For the ESN model, a random search strategy was used with the Hyperopt library to fine-tune key hyperparameters such as reservoir size, leak rate, input scaling, and spectral radius. Ridge regression was applied for regularization.

The performance of the models was evaluated in terms of Mean Square Error (MSE), Average Euclidean Distance Error (ADE), and Smoothness using the Spectral Arc Length (SPARC) metric:

- ESN: Achieved an MSE of 0.0614 m^2 and an ADE of 0.312 m, with a SPARC of 23.1.
- TCN: Recorded an MSE of 0.065 m^2 and an ADE of 0.309 m, with a SPARC of 25.09.
- Hybrid TCN-ESN: Outperformed both models with an MSE of 0.0565 m^2 and a SPARC of 19.39, offering a 13% improvement in accuracy compared to the TCN alone. The hybrid model's computational overhead was higher, requiring 2406 parameters, compared to 1002 for the ESN and 2034 for the TCN.

The results demonstrate that ESNs are ideal for real-time applications due to their minimal computational footprint, making them well-suited for IoT systems or low-power devices. While the hybrid TCN-ESN model provides the best accuracy and smoothness, its higher resource requirements may limit its deployment in resource-constrained environments. However, this model could be beneficial in applications like robotic navigation, augmented reality (AR), or autonomous vehicles, where accuracy and smooth trajectories are critical.

Despite the promising results, several challenges remain:

- Parameter Sensitivity: ESNs are highly sensitive to hyperparameters, such as reservoir size, leak rate, and spectral radius, which require fine-tuning for optimal performance. Developing more robust ESN architectures that are less dependent on specific hyperparameters is an important area for future work.
- Computational Overhead: Although the hybrid TCN-ESN model achieved superior accuracy, its higher computational cost may limit its use in low-resource environments. Future efforts could focus on network pruning or quantization to reduce complexity while maintaining performance.

Future work should also explore the integration of multimodal sensor data—such as Wi-Fi signals, vision-based inputs, and other sensor types—along with the capacitive sensors to improve robustness. Additionally, exploring feedback loops, cascaded reservoirs, and separate reservoirs for predicting x and y coordinates could offer new possibilities for enhanced localization systems.

This thesis presents an effective approach for indoor localization using ESNs, TCNs, and capacitive sensors. The research demonstrates the ability of ESNs to provide real-time localization with minimal computational overhead, while TCNs offer robust modeling of long-term temporal patterns. The hybrid TCN-ESN model represents a promising solution for achieving high accuracy and smooth trajectory predictions in indoor localization tasks. Future research should focus on addressing the challenges of parameter sensitivity and computational overhead to further enhance the practical applicability of these models in real-world environments.

Acknowledgement

I would like to express my deepest gratitude to those who have supported and guided me throughout the journey of completing this thesis.

First and foremost, I want to thank my parents for their unwavering love, encouragement, and support. Their belief in me has been a constant source of motivation.

I am profoundly grateful to my fiancée, Mahsa, whose patience, understanding, and support have been invaluable. Her encouragement and love have been a cornerstone of my strength.

I would also like to extend my sincere thanks to my professor, Mihai Teodor Lazarescu, for his insightful guidance, expertise, and continuous support. His mentorship has been instrumental in shaping this research and enhancing my academic growth.

Finally, I am thankful to Giorgia for her assistance and help throughout this process. Her contributions have significantly enriched my work.

Thank you all for your encouragement and support.

Table of Contents

Table of figures

1. Introduction

1.1 Background on indoor localization techniques

Localization and identification technologies are critical components in various fields, including robotics, smart environments, and security systems. Localization refers to determining the position of an object or person within a certain space, while identification involves recognizing and differentiating between various entities based on specific characteristics or data.

Localization systems can be categorized into active and passive methods, each with distinct characteristics and applications.

Active localization systems require the emission of signals from an external source to determine the position of an object or person. These systems use transmitters that send out signals which are then captured by receivers to calculate the location.

Several technologies are employed for active localization and identification, each with its advantages and limitations.

Global Positioning System (GPS): GPS is widely used for outdoor localization, providing accurate position information based on satellite signals. It is not suitable for indoor environments due to signal attenuation and multipath effects.

Radio Frequency Identification (RFID): RFID uses radio waves to identify objects with tags. It is commonly used for tracking items in logistics and retail. However, its accuracy can be limited by tag placement and interference (Obeidat, 2021).

Wi-Fi and Bluetooth Beacons: Wi-Fi and Bluetooth beacons can provide indoor localization by measuring signal strength or time-of-flight. These technologies are useful in smart homes and retail environments but can be affected by signal interference and varying propagation conditions.

Ultrasound and Infrared Systems: Ultrasound and infrared systems can offer high precision for indoor localization by measuring the time it takes for signals to travel between transmitters and receivers. They are less prone to signal interference but require a clear line of sight.

Active systems require the installation of additional equipment, such as transmitters or beacons, which can increase infrastructure costs. Also, active components require regular maintenance and power supply to ensure continuous operation. Active systems often require the user to wear an active tag or interact with the system, which can be inconvenient and may affect the overall user experience.

One the other hand, passive localization systems rely on naturally occurring signals or environmental interactions to determine the position, without requiring emissions from the target.

Therefore, there is no need for additional signal emitters, reducing installation and maintenance expenses. Also, these systems do not require the user to wear tags or interact with the system, making it less intrusive and easier to integrate into existing environments.

Among the most common tag less approaches are infrared sensing, which detects human body heat to produce thermal images (M. Kuki, 2013), ultrasound sensing that calculates the time-of-flight of sound waves (Bordoy, 2015), and radio detection and ranging (radar) (Y. Kim, 2015) . Although image-based systems can be effective, they are often expensive and raise privacy concerns.

On the other hand, long-range capacitive sensors operating in load mode utilize single-plate transducers, with the human body acting as the second conductive plate. These sensors are affordable, easy to conceal for aesthetic purposes, and maintain privacy. However, they are prone to environmental noise, which reduces the accuracy, stability, and range of localization. Improvements to their noise resistance can be achieved through better sensor plate design, enhanced electromagnetic fields, or advanced signal acquisition and processing techniques (G. Subbicini, 2023).

1.2 Importance of Accurate Indoor Positioning

Accurate indoor positioning is crucial for a wide range of applications, from enhancing user experiences in smart buildings to improving emergency response times. In environments like hospitals, airports, and shopping centers, precise indoor localization helps in guiding individuals, managing assets, and optimizing resource usage. It also plays a key role in energy efficiency by enabling systems to adjust lighting, heating, and cooling based on real-time occupancy data. Furthermore, accurate indoor positioning is vital in assisted living systems, providing valuable data for monitoring the well-being of elderly individuals, ensuring safety, and promoting independent living. In security applications, it allows for real-time tracking of unauthorized movements, enhancing surveillance and intrusion detection systems. Therefore, the objective of the thesis is to enhance the accuracy of indoor positioning. I decided to enhance the accuracy using echo state networks because it introduces little complexity to the system.

1.3 Overview of Capacitive Sensing for Localization

Capacitive sensing is a non-invasive technology widely used for indoor localization, leveraging the interaction between conductive objects and electric fields to determine the position of individuals. In capacitive sensing, a single-plate transducer is employed, with the human body acting as the second conductive plate in the system. This type of sensing is particularly suitable for indoor environments where minimal interference with users is desired. Capacitive sensors are cost-effective, relatively easy to install, and maintain privacy, as they do not require the capture of visual data like cameras or thermal sensors (G. Subbicini, 2023).

Capacitive sensors are advantageous in their ability to function without requiring users to carry or wear specific devices, a feature that distinguishes them from active localization systems like RFID or Bluetooth beacons. By analyzing the variations in the electric field caused by the presence and movement of a human body, these sensors can track a person's position and movement within a given space. This tag-less nature makes capacitive sensing ideal for applications in environments like homes, hospitals, and public buildings, where ease of use and unobtrusiveness are essential. Additionally, capacitive sensing is resilient to some common issues in other indoor positioning systems, such as line-of-sight obstruction and privacy concerns related to image-based systems (Y. Kim, 2015).

However, capacitive sensors also have certain limitations. One of the primary challenges is environmental noise, which can significantly impact the accuracy and stability of the system. Electromagnetic interference from other electronic devices, changes in humidity, and the presence of non-conductive objects in the room can all affect the quality of the signal. This noise sensitivity can reduce the precision of localization, making it difficult to maintain stable performance across different environments. To address these challenges, research is being conducted on improving sensor design and signal processing methods. (G. Subbicini, 2023) suggests that better sensor plate design, stronger electromagnetic fields, and advanced filtering techniques can improve the robustness of capacitive sensing systems.

Despite its limitations, capacitive sensing is a promising technology for indoor localization, especially in assisted living systems, smart homes, and security applications. It offers a relatively low-cost solution with the potential for non-intrusive, privacy-preserving tracking. Future advancements in sensor technology and noise resistance could further enhance the capabilities of capacitive sensing, making it a competitive alternative to other indoor positioning technologies, such as infrared, ultrasound, and radar-based systems (Bordoy, 2015).

1.4 Brief introduction on TCN and ESN

Echo State Networks (ESNs) are a type of recurrent neural network (RNN) that leverages a fixed, random, and sparse reservoir of neurons to capture temporal dynamics. Unlike traditional RNNs, where all weights are learned during training, the reservoir in an ESN remains unchanged after initialization. The main learning task in ESNs involves training only the output weights, which simplifies the training process and helps address the vanishing gradient problem commonly encountered in deep RNNs. ESNs are particularly well-suited for tasks involving time-series data and sequential information, as they can effectively model complex temporal dependencies without requiring extensive computational resources (Jaeger, 2001).

Temporal Convolutional Networks (TCNs), on the other hand, are a class of convolutional neural networks designed to handle sequence data by capturing long-term dependencies through dilated convolutions. TCNs use convolutional layers with dilated filters to expand the receptive field, allowing the network to capture information from a wide range of time steps without increasing the number of parameters significantly. This architecture enables TCNs to model sequences with long-term dependencies effectively, offering advantages over traditional RNNs and LSTMs, such as improved parallelism and reduced training times. TCNs have shown significant promise in applications involving sequence-to-sequence tasks, such as time-series forecasting and natural language processing (Shaojie Bai, 2018).

In summary, while ESNs are designed to capture temporal dynamics with a fixed reservoir and trainable output weights, TCNs utilize dilated convolutions to handle long-range dependencies in sequences. Both architectures offer unique advantages for modeling temporal data, with ESNs providing simplicity and efficiency, and TCNs offering powerful capabilities for capturing complex temporal patterns through convolutional layers.

1.5 Thesis Objectives and Research Questions

1.5.1 Thesis Objectives:

1. Evaluate the Performance of Echo State Networks (ESNs) for Indoor Localization:
   - 1.1. Investigate how ESNs can be effectively utilized for indoor positioning tasks using data from capacitive sensors.
   - 1.2. Assess the impact of different hyperparameters on the accuracy and computational efficiency of ESNs in indoor localization applications.
2. Compare ESNs with Temporal Convolutional Networks (TCNs):
   - a. Analyze and compare the performance of ESNs and TCNs in terms of localization accuracy and computational cost using the same dataset.
3. Integrate ESNs and TCNs for Improved Localization:
   - a. Develop a hybrid model that combines ESNs and TCNs to leverage the advantages of both architectures for indoor localization.
   - b. Evaluate the performance of the integrated model in terms of accuracy and computational efficiency compared to standalone ESN and TCN models.

1.5.2 Research Questions:

1. How effective are Echo State Networks (ESNs) in achieving accurate indoor localization using capacitive sensor data?
2. In what ways do Temporal Convolutional Networks (TCNs) outperform or underperform compared to ESNs in the context of indoor positioning?
3. What are the benefits and drawbacks of combining ESNs and TCNs into a hybrid model for indoor localization?
4. How does the integrated model compare to standalone ESNs and TCNs in terms of accuracy and computational cost?

These objectives and research questions provide a structured approach to investigating the capabilities of ESNs and TCNs for indoor localization, and they outline a comprehensive plan for evaluating and improving these models.

2. Literature Review

2.1 Indoor Localization Techniques

Localization and identification technologies are essential across various domains, such as robotics, smart environments, and security systems. Localization refers to determining the position of a person or object within a defined space, while identification involves distinguishing between different entities using specific characteristics or data.

These systems can be categorized into active and passive methods, each offering distinct features and applications. Active localization systems require external signal emissions to determine location. They rely on transmitters that send out signals, which are then captured by receivers to calculate the object or person's position. Different technologies are used for active localization, each with its own strengths and limitations. The most common active indoor localization techniques are the following.

Most common active methods are as follows:

Satellite-Based Navigation**:** The Global Positioning System (GPS) is the most used system for outdoor localization. However, it relies on the line-of-sight (LOS) between satellites and the device, which becomes inefficient for indoor location-based services due to interference from building walls. GPS performance can be enhanced by utilizing a steerable, high-gain directional antenna at the front end of the receiver (Nirjon, 2014). In areas where GPS signals are inaccessible, pseudolites (pseudo-satellites) are employed as standalone localization systems. These systems consist of pseudolites, transmission and receiver antennas, target receivers, and reference points. The main idea is to capture GPS signals and retransmit them via indoor transmitters (Xu, 2015).

Radio Frequency (RF)-based systems: RF-based systems are the most used for localization, largely due to their ability to cover large areas with low-cost hardware. RF waves can penetrate materials like walls and human bodies, making them more effective than other systems such as infrared (IR) and ultrasonic navigation. However, RF-based systems are not ideal for use in hospitals and airplanes due to the potential for interference with existing RF systems.

Wireless technologies for indoor localization are categorized based on the radio frequencies they operate on, typically below 300 GHz in the radio spectrum. The frequency of a wireless technology influences factors such as coverage, wall penetration, and resistance to obstacles. Consequently, wireless technologies are divided into three categories depending on their range: long-distance, mid-distance, and short-distance, each suited to different applications. Factors such as complexity, accuracy, and environment play key roles in determining which type of system is used for a specific purpose (Ahson, 2010).

In Wireless Sensor Networks (WSNs), knowing the position of nodes is crucial for functions like routing, clustering, and context-based applications. A WSN consists of nodes that sense environmental data (e.g., temperature, humidity, luminosity) and transmit the collected information wirelessly to a sink node for data collection. Examples of WSN applications include fire control systems, smart homes, and rescue operations (Alkhatib, 2011). WSN localization, which can use either range-based or range-free methods, relies on technologies like the ZigBee standard and RSSI (Received Signal Strength Indicator) for tracking the location of nodes. This process can involve precise methods such as trilateration or simpler approaches like proximity and scene analysis.

Common examples of RF-based navigation systems include WiFi, Bluetooth, Zigbee, Ultra-Wideband (UWB), and Radio Frequency Identification (RFID), each offering different advantages based on the specific use case (Obeidat H. S., 2021).

Sound-Based Technologies: Sound-based localization uses ultrasonic or acoustic waves to determine positions. Ultrasonic waves are effective over short distances and have low power requirements, while acoustic systems use sound captured by microphones for localization. Though cost-effective and reliable in certain environments, these systems can be affected by reflections and synchronization issues (Obeidat H. S., 2021).

Active systems require the installation of additional equipment, such as transmitters or beacons, which increases infrastructure costs. Additionally, these systems need regular maintenance and a continuous power supply to operate. They often require users to wear tags or actively interact with the system, which can be inconvenient and reduce user experience.

On the other hand, passive localization systems rely on natural signals or environmental interactions to determine the location, eliminating the need for signal emissions from the target. As a result, no extra signal emitters are needed, reducing installation and maintenance costs. Furthermore, users do not need to wear tags or interact with the system, making these systems less intrusive and easier to integrate into existing infrastructures. The most common passive methods are the following.

Most common passive methods are as follows:

Infrared (IR) Sensing: Among common tag-less approaches are infrared sensing, which detects body heat to create thermal images (M. Kuki, 2013).Passive Infrared (PIR) sensors detect changes in thermal radiation when a person moves within a sensor's field of view. IR sensing is commonly used in motion detection, smart lighting systems, and occupancy detection in smart homes and buildings. It is highly reliable in detecting movement and presence but is limited by its line-of-sight requirement and sensitivity to obstructions such as walls. The disadvantage of these systems is their limited range and susceptibility to interference from environmental heat sources.

Ultrasound sensing: Ultrasound systems measure the time it takes for sound waves to travel from a transmitter to an object and back to the receiver (time-of-flight). The human body reflects these sound waves, allowing the system to determine the person's location (Bordoy, 2015). Ultrasound is often used in indoor positioning systems for navigation, robotics, and safety applications. It can function in low-light environments and is not affected by visual obstructions like smoke or dust. Ultrasound systems require careful placement and calibration to ensure accuracy and are limited by their shorter range.

Radio detection and ranging (radar): Radar systems use radio waves to detect and track objects by analyzing the reflections of the waves from the target. In indoor localization, radar can detect human movements and positions by measuring the time it takes for radio signals to reflect back from the human body (Y. Kim, 2015). Radar is used in security systems, human activity monitoring, and intrusion detection. It can penetrate walls and detect movement in non-line-of-sight situations. High cost and complex setup are the disadvantages of these systems.

Capacitive sensors: It offers a promising passive localization solution. Long-range capacitive sensors in load mode use single-plate transducers, with the human body acting as a second conductive plate. These sensors are cost-effective, easy to conceal for aesthetic reasons, and ensure privacy. However, they are sensitive to environmental noise, which can affect their accuracy, stability, and range. Noise resistance can be improved with better sensor design, enhanced electromagnetic fields, or advanced signal processing techniques (G. Subbicini, 2023). Capacitive sensing is used in smart homes, assisted living systems, and for security purposes due to its low cost, privacy-preserving nature, and ease of concealment. Low cost, easy integration, and privacy-friendly (no cameras or tracking devices required) are among their advantages. In this thesis the focus is localization using the data collected by this type of sensor.

2.2 Capacitive Sensing in Human Localization

Capacitive sensing technology is widely utilized for human localization, where it detects the presence and movement of individuals by measuring changes in capacitance between a sensor and a human body. This technology has gained significant attention in various fields such as smart homes, healthcare, and security systems due to its low-cost implementation, low power consumption, and the ability to maintain user privacy. Capacitive sensors work by creating an electric field between two conductive plates, with the human body acting as one of the plates. Changes in this field, caused by movement or proximity of a person, are used to estimate their location or interaction with the environment (Tobias Grosse-Puppendahl, 2017).

Capacitive sensing techniques can be categorized in four different operating modes: loading, shunt, transmit, and receive. The first three modes; (Shunt, Transmit, and Receive); require at least two separate capacitor plates, which can increase both the cost and complexity of the system. In contrast, the Loading mode simplifies the setup by using a single electrode for both transmitting and receiving signals. Additionally, all modes except Loading can operate in either active or passive sensing mechanisms.

In active capacitive sensing, a known signal is applied to the transmit electrode, which then interacts with the body part through capacitive coupling. This interaction results in a signal being transferred to the receive electrode. By measuring the strength of the signal at the receive electrode, the presence and movement of the body part can be detected.

In contrast, passive capacitive sensing does not generate its own signal. Instead, it relies on existing external or ambient electric fields. These external fields are detected by the system without the need for additional signal generation (Raphael Wimmer, 2007).

In Load mode operation, only a single sensor plate is required, with the tracked person acting as the other plate of the capacitor. This configuration simplifies and reduces the cost of deployment. However, the sensitivity of this setup depends on the size of the sensor plate used. A larger plate typically results in higher sensitivity. Figure 2.1 shows the load operating mode.

Researchers at Polito have developed a system that enhances sensitivity by using a load mode-operated transducer along with advanced data acquisition techniques. This system effectively addresses key constraints for indoor human localization, including low power consumption, cost efficiency, tag-less operation, and privacy considerations (Ramezani Akhmareh, 2016).

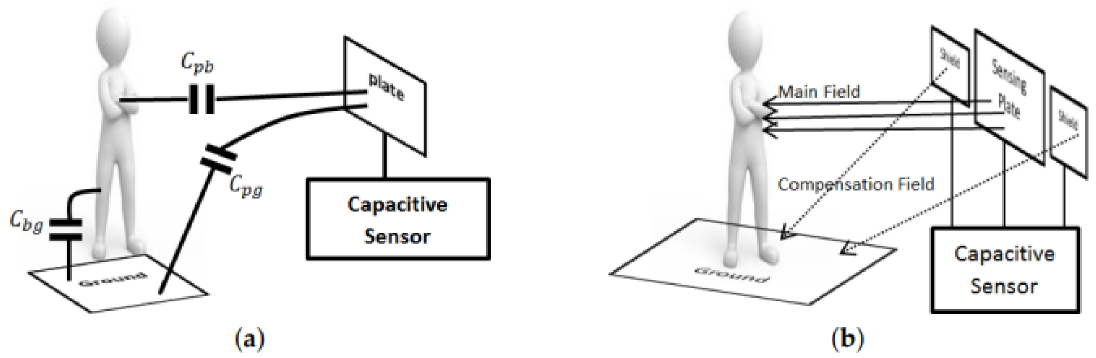

Figure 2.1. (a) Main sensor capacitances in load mode: plate-body (C_{pb}), plate-ground (C_{pg}), and body-ground (C_{bg}). (b) Use of compensation fields for short range load mode capacitive sensors to reduce C_{pg}.

In this thesis the data collected from this system has been used to develop the localization system.

2.3 Echo State Networks

2.3.1 Architecture and principles

In recent years, the field of machine learning has seen remarkable advancements in the modeling and prediction of complex temporal and sequential data. Among various models, Echo State Networks (ESNs), a subset of reservoir computing, have emerged as a robust and efficient approach for handling dynamic systems and time-series prediction. This chapter provides a comprehensive exploration of Echo State Networks, focusing on their theoretical foundations, architectural components, training methodologies, and practical applications.

Echo State Networks are distinguished by their unique approach to recurrent neural networks (RNNs). Unlike traditional RNNs, which require extensive training of both the recurrent and output weights, ESNs simplify the learning process by fixing the weights of the recurrent connections within the reservoir. This approach not only reduces computational complexity but also enhances the model's capability to capture temporal dependencies in sequential data.

The chapter begins with an overview of Reservoir Computing, the broader framework within which ESNs are situated. This section lays the groundwork for understanding the principles of ESNs by explaining the concept of the reservoir, a crucial component that generates a rich set of internal states in response to input data. Following this, the chapter delves into the architecture of Echo State Networks, detailing the roles of the input layer, reservoir, and output layer. Key concepts such as the spectral radius and leaking rate, which influence the reservoir's dynamics, are discussed to highlight their importance in maintaining the Echo State Property (ESP).

Training methods for ESNs are covered in depth, focusing on the generation of reservoir states and the training of output weights. The chapter explains how these methods leverage linear regression techniques, such as ridge regression, to optimize the output weights while keeping the reservoir dynamics fixed.

In conclusion, this chapter aims to provide a thorough understanding of Echo State Networks, equipping readers with the knowledge needed to apply ESNs effectively in the context of indoor localization and other related applications. By examining the theoretical background, architectural details, and practical considerations, this chapter sets the stage for the subsequent analysis and implementation of ESNs in the thesis.

Reservoir Computing and Echo State Networks:

Reservoir Computing (RC) is a type of Recurrent Neural Network (RNN) that separates the model into two distinct parts: the reservoir and the readout layer.

1. The Reservoir: This part consists of a large, randomly connected network of hidden units that are not trained. The connections between these units are sparse and the reservoir's

behavior is dynamic, meaning it processes inputs in a complex, evolving manner. Once set up, the reservoir remains fixed and does not undergo training.

2. The Readout Layer: This layer is typically a simple, linear network that takes the processed information from the reservoir and makes predictions or decisions. The readout layer is trained using straightforward and efficient methods, similar to those used in feed-forward neural networks.

Reservoir Computing is notable for its simplicity in training and is also considered to have strong biological parallels, meaning it mimics some aspects of how biological systems process information.

Reservoir networks handle sequence processing tasks by separating the system into two components: a fixed, dynamic reservoir and an adaptive readout layer. The reservoir captures and maintains the temporal dynamics of the input sequence without requiring training. Meanwhile, the readout layer, which is trained, performs the task of mapping these dynamics to the desired output.

The key advantage of reservoirs is that they can effectively distinguish between different input sequences even without training, as long as they meet a few straightforward criteria. This means that the training process can be simplified to adjusting the weights of the readout layer alone, bypassing the need for complex recurrent weight adjustments. This separation results in a more efficient design for recurrent neural networks (RNNs) (Gallicchio, 2011).

Reservoir Computing (RC) includes several types of Recurrent Neural Network (RNN) models. Among these are well-known models such as Echo State Networks (ESNs) (Jaeger, 2001) and Liquid State Machines (LSMs) (W. Maass, 2002). This thesis specifically focuses on the Echo State Network (ESN) model.

An Echo State Network (ESN) is composed of three main components:

1. Input Layer: Consists of N_u units that process the input data.
2. Reservoir: This is a large collection of N_R sparsely connected recurrent hidden units.
3. Output Layer: Composed of N_Y units, typically linear and non-recurrent, which make up the readout layer.

In an ESN, the reservoir is responsible for encoding the input sequence into a high-dimensional space, while the readout layer maps this encoded information to the output. The basic equations describing the computation carried out by an ESN are as follow:

$$x(t) = f\left(W_{in}u(t) + \widehat{W}x(t - 1)\right) \tag{1}$$

$$y(t) = W_{out}x(t) \tag{2}$$

Where, $W_{in} \in R^{N_R \times N_u}$, is the input to reservoir weight matrix. $\widehat{W} \in R^{N_R \times N_R}$ is the recurrent reservoir weight matrix, and $W_{out} \in R^{N_Y \times N_R}$ is the reservoir to output weight matrix. The basic architecture of an standard ESN has been shown in Figure 2.2 (Gallicchio, 2011).

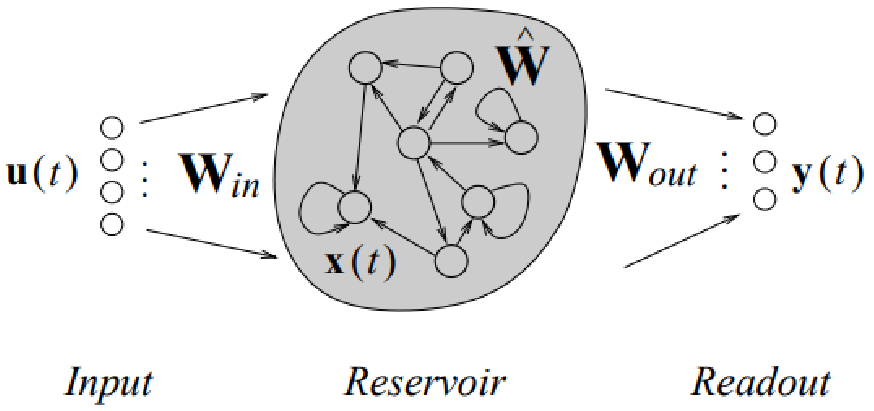

Figure 2.2 The architecture of an ESN

In Echo State Networks (ESNs), the activation function commonly used for the reservoir units is the hyperbolic tangent (tanh). This function helps the network handle non-linear transformations of the input data by squashing the values within the range of -1 to 1, making the network's dynamic responses more stable.

On the output side of the network, the result is obtained as a linear combination of the reservoir's activations. This means that the output layer does not apply any non-linearity but instead combines the weighted activations of the reservoir units to produce the final prediction or classification. This approach simplifies training by reducing the complexity of learning to a linear regression problem, while the reservoir handles the non-linear dynamics of the input sequence.

Echo state property:

Not every configuration of the input weights W_{in} and the reservoir weights $\widehat{W}$ results in a valid Echo State Network (ESN). For the ESN to work effectively, it must meet certain conditions that ensure the echo state property.

A valid Echo State Network (ESN) satisfies the Echo State Property (ESP), which is fundamental to its operation. The ESP ensures that after being driven by a long input sequence, the network's state depends solely on that sequence, rather than on its initial state. This means that the network gradually "forgets" its initial conditions as it processes more data. Essentially, the network's current state, denoted as x(n), becomes a function of its past input history, regardless of where the network started (Jaeger, 2001).

In other words, the ESN's state evolves in such a way that the influence of the initial state diminishes as the input sequence grows longer. This property is crucial because it guarantees that the network is responsive to recent inputs and that the effects of any arbitrary starting condition do not interfere with its predictive capabilities. The ESP is one of the key features that make ESNs effective for temporal data processing and sequence prediction tasks (Gallicchio, 2011).

In Jaeger (2001), two conditions are outlined as necessary and sufficient for an Echo State Network (ESN) with a hyperbolic tangent (tanh) activation function to have echo states:

Necessary Condition: The spectral radius (the largest absolute value of the eigenvalues) of the reservoir's recurrent weight matrix $\widehat{W}$ must be less than one. This ensures that the activations in the reservoir do not grow uncontrollably, preventing instability in the network's dynamics.

$$\rho(\widehat{W}) < 1 \tag{3}$$

Sufficient condition: The sufficient condition for ensuring the presence of echo states in an Echo State Network (ESN) is that the largest singular value of the reservoir's recurrent weight matrix $\widehat{W}$ is less than one. This condition guarantees that the dynamical system within the reservoir will not amplify input signals uncontrollably over time, leading to stable internal representations of the input sequence.

$$\sigma(\widehat{W}) < 1 \qquad (4)$$

In practical applications of Echo State Networks (ESNs), it is common to check only the necessary condition. While the sufficient condition guarantees the presence of echo states, it is often considered too restrictive.

Initialization and Training of ESNs

To create a functional Echo State Network (ESN), we can start with randomly generated matrices for the input-to-reservoir (W_{in}) and recurrent weights ($\widehat{W}$). These matrices are generally initialized with values drawn from a uniform distribution within a symmetric range. For Win, an input scaling parameter, w_{in}, is used so that the input weights fall within the interval $[-w_{in}, w_{in}]$. The recurrent weight matrix $\widehat{W}$ is then adjusted to meet the required conditions. Although this process doesn't guarantee echo states, most ESN literature advises scaling the spectral radius of $\widehat{W}$ to satisfy the necessary conditions. To adjust a randomly initialized reservoir recurrent matrix ($\widehat{W}_{random}$), you scale it to obtain the final matrix $\widehat{W}$:

$$\widehat{W} = \frac{\rho}{\rho(\widehat{W}_{random})}\widehat{W}_{random} \qquad (5)$$

Here, ρ represents the target spectral radius for the matrix $\widehat{W}$. After scaling, the spectral radius of $\widehat{W}$ will match the desired value ρ. In practice, values of ρ near 1 are frequently used because they push the reservoir's dynamics close to the edge of chaos (W. Maass, 2002), which often leads to better performance in applications (Jaeger, 2001).

Echo state network hyperparameters

In Echo State Networks (ESNs), several hyperparameters play a crucial role in determining the network's performance. Here are some key hyperparameters:

1. Spectral Radius: This controls the amplification of the network's dynamics. Values close to 1 are typically used to keep the dynamics near the edge of chaos, which is often beneficial for performance.

2. Input Scaling: This parameter scales the input weights in the input-to-reservoir matrix. It affects how the input signals are transformed and fed into the reservoir.

3. Reservoir Size: The number of neurons or units in the reservoir. A larger reservoir can capture more complex dynamics but also increases computational cost. Larger reservoir can also lead to overfitting if not managed properly.

4. Leak Rate: This parameter determines how quickly the states of the reservoir decay over time. It affects the memory of the reservoir and can influence performance depending on the task.

5. Regularization Parameter (λ): In ridge regression, this parameter controls the amount of regularization applied. A higher λ increases the penalty on the size of the coefficients, leading to greater regularization and potentially reducing overfitting. Conversely, a lower λ allows the model to fit the training data more closely, which could increase the risk of overfitting.

2.3.2 Applications in Time Series Prediction and Localization

Echo State Networks (ESNs) have shown significant promise in the field of time series prediction due to their ability to model complex temporal dependencies. Unlike traditional recurrent neural networks, ESNs utilize a fixed, random recurrent matrix, which simplifies training and allows for efficient handling of time-dependent data. This architecture is particularly well-suited for predicting future values in a time series by leveraging the reservoir's dynamic responses to past inputs. Research has demonstrated the effectiveness of ESNs in various domains, including financial forecasting, weather prediction, and demand forecasting. For example, (Jaeger, 2001) highlighted the advantages of ESNs in time series prediction tasks due to their ability to capture and utilize temporal patterns without the need for extensive parameter tuning.

In addition, ESNs have emerged as an efficient tool for time series prediction, particularly for complex, nonlinear systems. The architecture of an ESN consists of a large, fixed, and randomly connected recurrent neural network called a reservoir, which captures the dynamics of the input data. This reservoir of neurons is only partially trained, while the primary learning occurs in the output layer. This makes ESNs computationally efficient compared to traditional recurrent neural networks, where all weights are trained. The reservoir's ability to retain dynamic memory makes it particularly effective in predicting chaotic systems, such as the Mackey-Glass system, where ESNs have demonstrated significant improvements in prediction accuracy (Herbert Jaeger, 2004).

2.4 Temporal Convolutional Networks

2.4.1 Architecture and Principles

Temporal Convolutional Networks (TCNs) are designed to process sequential data and are particularly useful for tasks involving time series or sequence modeling. The core architectural features of TCNs are:

- 1D Convolutional Layers: TCNs apply convolutions across temporal sequences, meaning the network processes data sequentially while maintaining the order of the input. Unlike standard Recurrent Neural Networks (RNNs), which rely on recurrent connections, TCNs use convolutional filters to capture temporal dependencies.

- Causal Convolutions: In TCNs, causal convolutions ensure that the network processes data in a strictly time-ordered manner. This means that the output at a given time step is generated using only the current and previous time steps, preventing the "future" data from influencing predictions at earlier time steps.
- Dilated Convolutions: TCNs use dilated convolutions, which allow the network to handle long-range dependencies without increasing the computational complexity. Dilations expand the receptive field of the convolutional layers, enabling the network to learn from both local and distant time steps in a sequence.
- Residual Connections: TCNs incorporate residual or skip connections between layers. These connections help prevent the vanishing gradient problem and ensure that deeper layers continue to learn efficiently by allowing the network to bypass layers, improving the flow of gradients during backpropagation.

One key advantage of TCNs over traditional RNNs is that they allow for parallel processing due to the absence of recurrent connections, making them computationally efficient. Additionally, TCNs can manage sequences of arbitrary length through flexible receptive fields, which can be adjusted by modifying dilation rates and kernel sizes.

TCNs utilize multiple layers of dilated convolutions, which exponentially increase the dilation to capture broader input ranges more efficiently. These layers are combined with normalization, non-linear activation, and dropout for regularization, forming residual blocks with connections that help mitigate the issues associated with deep learning architectures. Dilated causal convolutional blocks of a temporal convolutional network has been shown in Figure 2.3. As it can be seen, it has an input tensor of input_size length, which has been repeated input_channel times. Each dilated convolutional block has nb_layers with nb_filters of kernel_size (purple arrows). The dilation factor increases exponentially along the hidde layers (G. Subbicini, 2023).

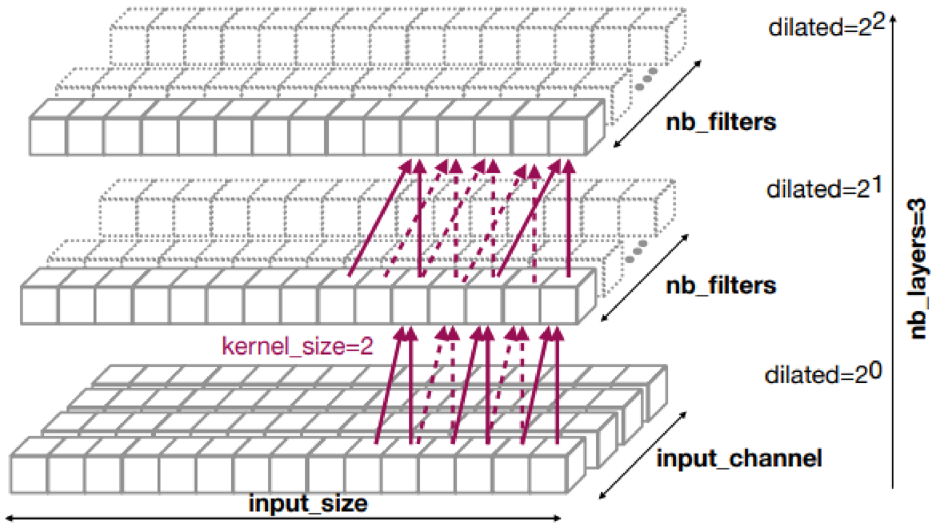

Figure 2.3 Dilated causal convolutional blocks of a TCN.

2.4.2 Applications in Sequence Modeling and Localization

TCNs are particularly well-suited for sequence modeling tasks that involve time series prediction, signal processing, and data with strong temporal dependencies. Their convolutional structure allows them to capture both short- and long-term patterns in sequential data, making them useful in fields such as natural language processing (NLP), audio signal processing, and financial forecasting. (Shaojie Bai, 2018) cover the various applications of TCNs in sequence modeling.

In the field of localization, TCNs have been applied to indoor positioning and tracking systems, where temporal sequences of sensor data (e.g., Wi-Fi signals, inertial measurement unit data) need to be processed to estimate a person's or object's location over time. By leveraging dilated convolutions, TCNs can capture complex temporal dynamics that reflect the movement of individuals or objects within a space, leading to more accurate and real-time localization (Lea, 2017).

Also, (G. Subbicini, 2023) shows that TCNs are well-suited for processing noisy data from capacitive sensors, as they can reject environmental electromagnetic noise through their ability to preserve resolution across layers. Experimental results show that TCNs outperform architectures like 1D-CNNs in terms of both inference accuracy and resource consumption when applied to indoor human tracking. TCNs achieved 12.7% lower inference loss with 73.3% less resource consumption compared to 1D-CNNs.

2.5 Integration of Different Neural Network Architectures for Improved Performance

Integrating different neural network architectures has become a prominent approach for enhancing the performance of machine learning models, particularly in complex tasks like time series prediction and localization. Combining the strengths of multiple models can lead to a system that compensates for the weaknesses of individual architectures. For instance, Convolutional Neural Networks (CNNs) excel at feature extraction from spatial data, while Recurrent Neural Networks (RNNs) are adept at modeling sequential dependencies. Hybrid models that combine CNNs with Long Short-Term Memory (LSTM) networks have been shown to effectively process both spatial and temporal data, enabling superior performance in applications like video analysis and speech recognition (T. N. Sainath, 2015). These models leverage CNNs to extract spatial features, which are then processed by LSTM layers to capture temporal patterns, thus enhancing both accuracy and generalization.

Moreover, the integration of Graph Neural Networks (GNNs) with recurrent and convolutional models presents new possibilities for improving the performance of tasks that involve structured data, such as sensor networks or social graphs. GNNs are effective at capturing relationships in graph-structured data, while architectures like RNNs or TCNs handle temporal aspects of the data. For example, in human activity recognition and indoor localization, combining GNNs with TCNs allows the model to capture spatial relationships between sensors while processing temporal sequences from sensor data streams. This integration leads to more accurate predictions, as it simultaneously models spatial dependencies across sensors and temporal dynamics in human movement (Wu et al., 2020). Hybrid models like these are becoming increasingly important in fields such as autonomous driving, robotics, and smart home systems, where the combination of spatial and temporal information is crucial for real-time decision-making.

(Alec Radford, 2016) presents an innovative approach to image generation by integrating Convolutional Neural Networks (CNNs) with Generative Adversarial Networks (GANs). In this framework, the generator and discriminator networks of the GAN are built using CNN architectures. This integration leverages the strength of CNNs in extracting and representing high-level features from images, resulting in significantly improved image quality and realism in the generated outputs. The use of CNNs enhances both the generator's ability to produce more detailed and coherent images and the discriminator's capacity to better evaluate the authenticity of generated images, leading to more effective unsupervised representation learning.

In this thesis integration of TCN and ESN for improving the performance of indoor localization has been exploited.

3. Methodology

3.1 Experimental Setup

3.1.1 Capacitive Sensor Operation and Deployment

The system employs four capacitive sensors, each equipped with a sensing plate measuring 16 cm × 16 cm. These sensors are mounted at chest level, positioned at the center of the "walls" within the virtual room (as depicted in Figure 3.1).

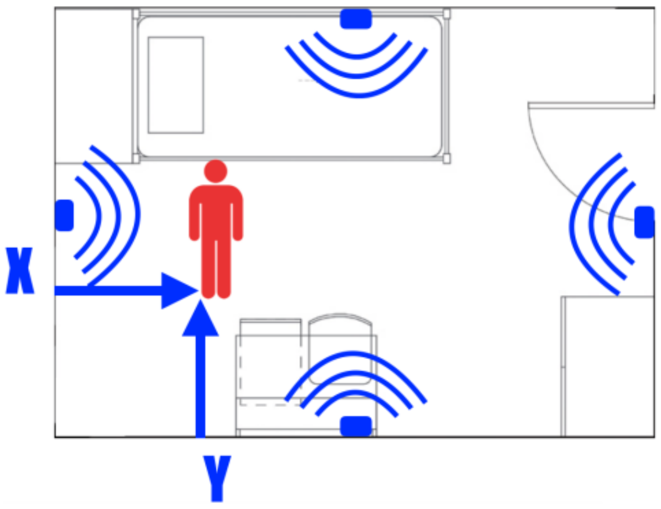

Figure 3.1 Capacitive sensors deployment in the 3 m × 3 m room.

The capacitive sensors operate with a transducer in load mode, where the capacitance changes according to the distance from a nearby human body or other objects. To mitigate the impact of these other objects, various signal processing techniques were applied (Ramezani Akhmareh, 2016).

3.1.2 Data Collection and preparation

The data is sourced from four single-plate load-mode capacitive sensors, which are positioned at the center of the virtual walls in a 3 m × 3 m laboratory space. These sensors are labeled with the coordinates of the person and record data at a rate of 3 Hz. The data undergoes preprocessing that includes a median filter with a 50-second input window, followed by a low-pass filter with a transition band between 0.3 and 0.4 Hz. The dataset is split into three parts: 60% for training, 20% for validation, and 20% for testing (for TCN) (Figure 3.2), with each part arranged in chronological order (G. Subbicini, 2023). For the ESN; as ridge regression is used; for training the readout layer the data is split into two parts: 80% for training and 20% for testing.

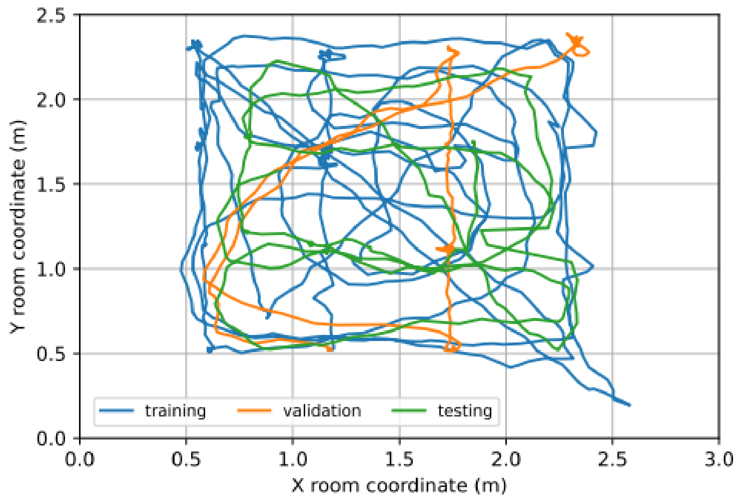

Figure 3.2 Data split for TCN model

3.2 Echo State Network Implementation

3.2.1 Architecture Design

First, I used a stand-alone ESN to predict the x, y values from received sensor values. The architecture of the network is as Figure 3.3.

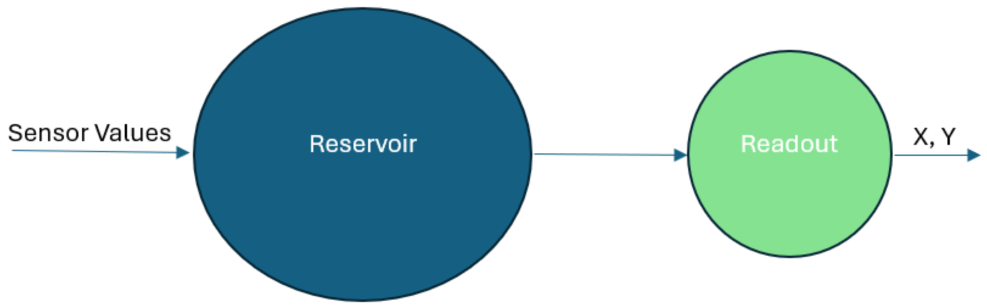

Figure 3.3 The architecture of ESN model.

3.2.2 Effect of Hyperparameter on Reservoir States

First, I investigated how these hyperparameters affect the behavior of Echo State Network: Spectral radius, Leaking rate, Input scaling

Spectral radius: The spectral radius is the largest absolute value of the eigenvalues of the reservoir's recurrent weight matrix $\widehat{W}$. The necessary condition for the ESN to meet Echo State Property is that the Spectral radius must be less than 1.

Here we see how different values of spectral radius affect the state of the neurons in the reservoir layer. I used the values of 4 sensor values as input of the reservoir and set a random seed for reproducibility. I checked the states of reservoir for three different values for spectral radius {0.1, 1.25, 10}

In Figure 3.4 the states of the 20 neurons of the reservoir have been plotted.

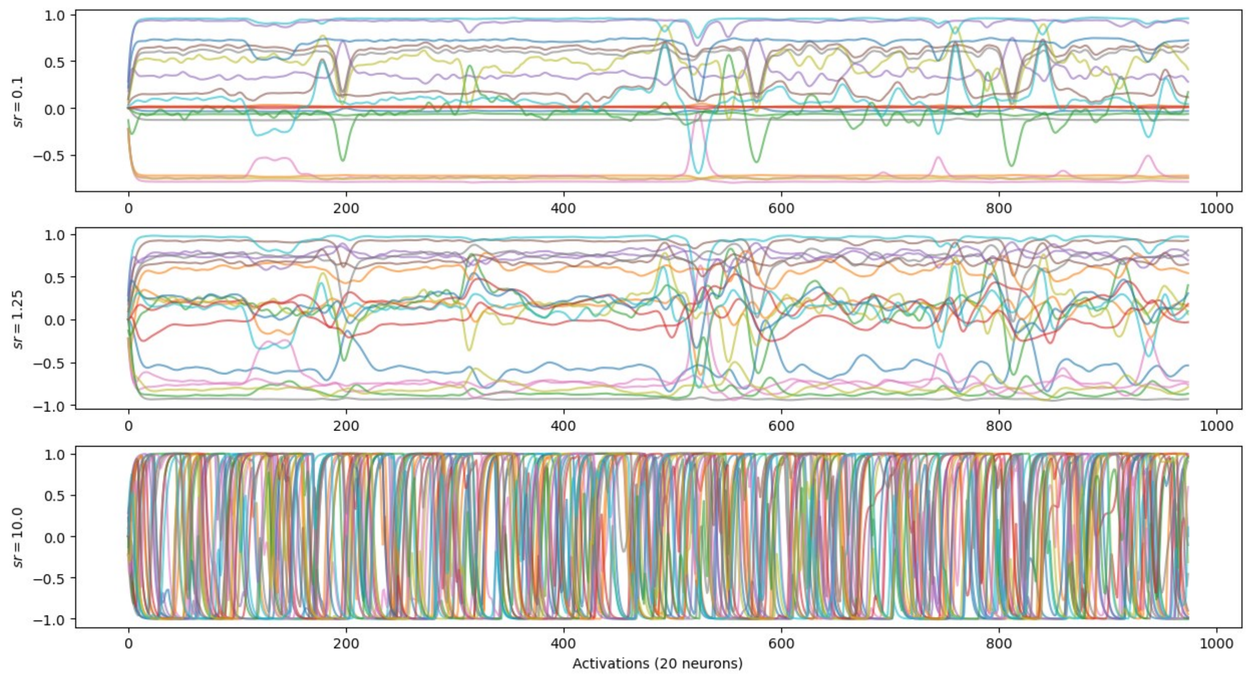

Figure 3.4 Impact of spectral radius on states of reservoir neurons

As it can be seen a lower spectral radius leads to more stable dynamics, while a higher spectral radius results in chaotic behavior.

A spectral radius close to 1 is theoretically believed to help the reservoir states be less influenced by their initial conditions while maintaining good memory properties. In practice, using a random search algorithm is often the most reliable method to determine the optimal spectral radius for a specific task (Gallicchio, 2011).

Input scaling: The input scaling is a coefficient applied to W_{in} that adjusts the gain of the inputs to the reservoir.

I have tried three different values {0.1, 1.0, 10} to see how it affects the reservoir state.

The result has been shown in Figure 3.5.

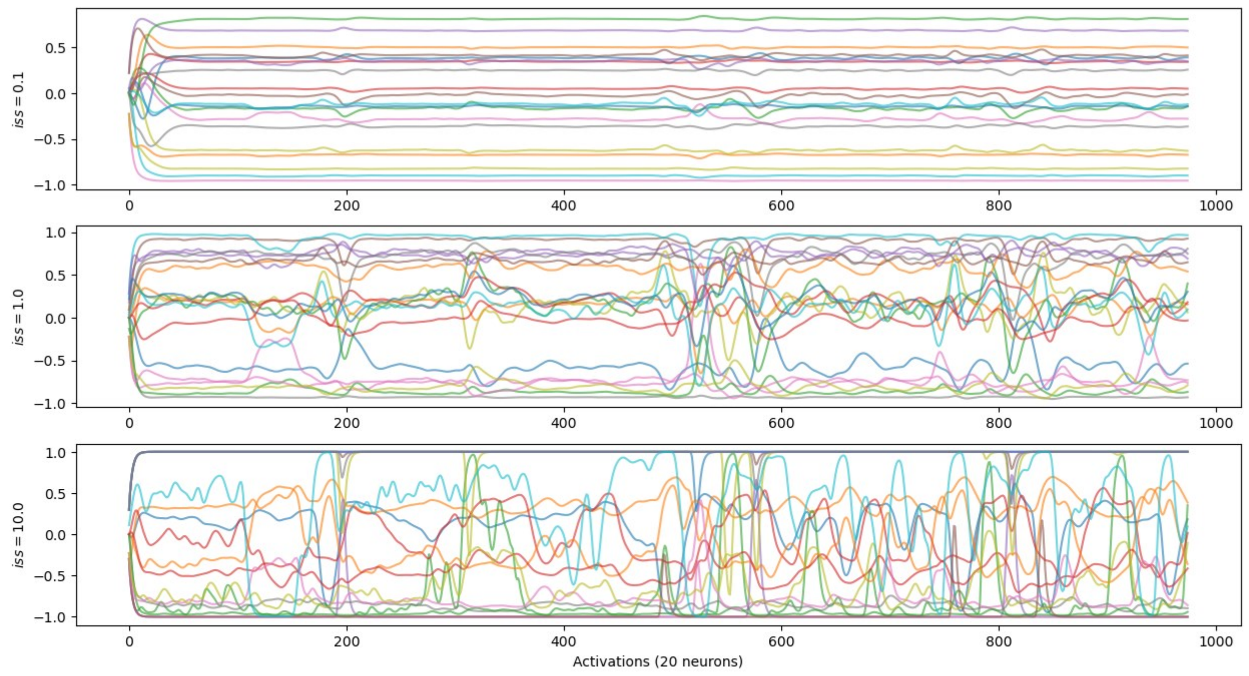

Figure 3.5 Impact of input scaling on states of reservoir neurons

The average correlation between reservoir states and inputs has been shown in Table 3.1

Table 3.1 correlation between input scaling and reservoir state

input scaling	correlation
0.1	0.0797
1	0.2089
10	0.2452

Increasing the input scaling typically enhances the correlation between the reservoir states and the input data.

This correlation increases with higher input scaling up to a certain level. Beyond this saturation point, further increases in input scaling may not significantly improve the correlation.

With lower input scaling, the reservoir states tend to be less influenced by the input signals and are more reflective of the reservoir's internal dynamics.

Leaking rate: The leaking rate in an Echo State Network (ESN) regulates how quickly the reservoir's state responds to new inputs versus maintaining its previous state. Essentially, it determines the time constant of the ESN, influencing how much past inputs affect the current state.

Figure 3.6 shows the effect of different leaking rates {0.02, 0.3, 1.0} on the states of reservoir.

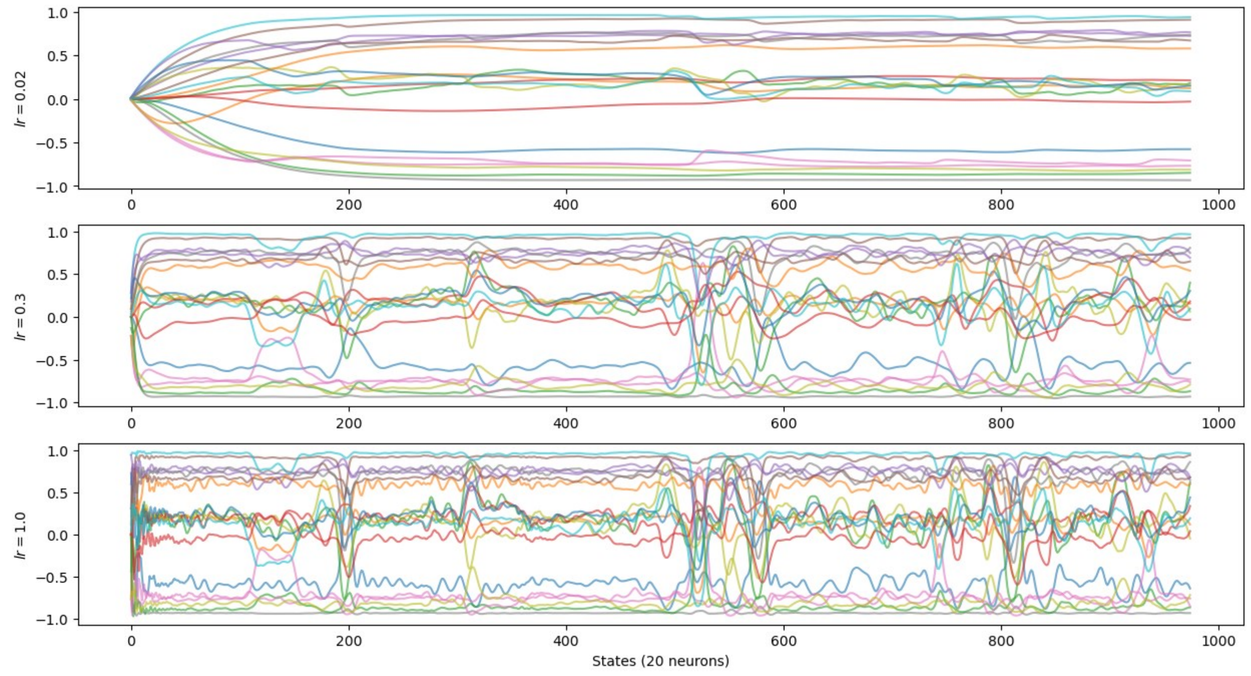

Figure 3.6 Impact of leaking rate on states of reservoir neurons

When the leaking rate is high, the reservoir states update more rapidly, making the network's response to recent inputs more immediate. This typically results in a shorter memory of past inputs, as the reservoir quickly adjusts to new information. It can lead to more responsive dynamics but might also make the system more sensitive to noise.

On the other hand, a low leaking rate causes the reservoir states to change more slowly, giving more weight to past inputs. This results in a longer memory of past inputs, as the network retains information over a longer period. It can improve stability and memory but might slow down the response to new inputs.

In summary, the leaking rate helps balance the trade-off between responsiveness and stability in the ESN.

3.2.3 Training and Evaluation Procedures

Training and evaluating an Echo State Network (ESN) involves several key steps, which include setting up the network architecture, training it, and evaluating its performance.

The first step is to define the ESN Architecture. It includes setting up the reservoir. For setting up the reservoir, reservoir size, spectral radius, leak rate, and input scaling should be defined. Then we should define the readout layer and a learning algorithm to be used for that layer. I used ridge regression as the learning algorithm for my readout layer. The regularization parameter (λ) should be selected for this layer.

After defining the architecture, and setting the reservoir size, spectral radius, leak rate, and input scaling the reservoir weights are initialized randomly.

Then, the input is fed into the network, allowing the reservoir to generate activations (states) based on the input data. The reservoir states are updated based on the input data and the previous state. The update equation is as follows:

$$x(t) = (1 - leak_{rate})x(t-1) + (leak_{rate})\tanh\left(W_{in}u(t) + W_{res}x(t-1)\right) \qquad (6)$$

where $x(t)$ is the reservoir state at time t, $u(t)$ is the input at time t, W_{in} is the input weight matrix, W_{res} is the reservoir weight matrix, and tanh is the activation function.

Then the reservoir states are collected during the feed-forward phase to train the output weights. To train the readout layer ridge regression is used. Ridge regression adds a penalty term to the least squares objective function, which helps to regularize the solution. The objective function for ridge regression is:

$$Objective = \left|\left|Y_{train} - X_{res}.W_{out}\right|\right|^2 + \lambda\left|\left|W_{out}\right|\right|^2 \qquad (7)$$

The output weight is calculated using the following formula:

$$W_{out} = (X_{res}^T X_{res} + \lambda I)^{-1} X_{res}^T Y_{train} \qquad (8)$$

Where, X_{res}^{T} is the transpose of reservoir weight matrix, I is the identity matrix with the same number of rows as X_{res} and λ is the regularization parameter, controlling the strength of the penalty term.

For evaluation of ESN test data is fed into the trained ESN to generate reservoir states using the test inputs. Then the predictions from reservoir states are computed using the learned output weights. Then, the predictions are compared with the actual test outputs using appropriate performance metrics. In this thesis, mean square error (MSE) is used as the performance metric.

3.2.4 Hyperparameter Tuning:

Hyperparameter tuning is to adjust hyperparameters such as reservoir size, spectral radius, input scaling, and leak rate to optimize the model performance. This can be done using techniques like grid search, random search, or more advanced methods such as Hyperopt which has been used in this thesis. Hyperparameter tuning of an Echo State Network (ESN) using Hyperopt involves defining the ESN model, specifying a search space for the hyperparameters, and using Hyperopt to find the optimal set of hyperparameters.

For creating the search space following values for each hyperparameter was selected:

Reservoir size: [10, 50, 100, 200, 400, 800, 1600]

Spectral radius: [0.3, 0.6, 0.9, 1.2]

Leak rate: [0.1, 0.2, 0.3, 0.5, 0.7, 0.9]

Input scaling: [0.5, 1]

Regularization parameter: [1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e-0, 1e1]

The Hyperopt selected one combination of these values in each run and initialized 100 ESN for the selected combination as the initial reservoir weight matrix has an effect to the result. As an evaluation metric I selected the mean square error (MSE). 1000 different combinations have been used for every reservoir size. Then the whole procedure repeated several times with shrunk intervals around the best-found solution.

The comparison between the different reservoir size and the test loss has been depicted in Figure 3.7. As it can be seen the best reservoir size is 500 for our dataset (loss = 0.0614 m^2).

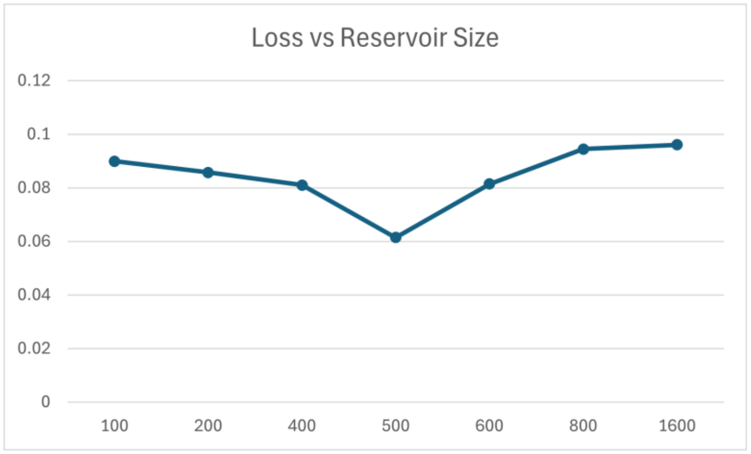

Figure 3.7 relationship between reservoir size and test loss

The best hyperparameters for every reservoir size and their corresponding test loss have been mentioned in Table 3.2.

Table 3.2. Best hyperparameters for different reservoir sizes for standalone ESN.

Reservoir Size	Leak rate	Spectral radius	scaling	ridge	Loss (m^2)
100	0.602	1.0422	0.6381	0.2133	0.0899
200	0.8718	1.3079	0.5134	0.3519	0.0857
400	1.0648	1.3187	1.4623	23.193	0.081
500	**1.070864**	**1.527258**	**2.006**	**16.0788**	**0.0614**
600	1.120874	1.454852	1.902	31.667	0.0814
800	1.0709	1.100513	1	9.996187	0.0944
1600	0.8637	1.1272	0.9436	15.6809	0.096

Figure 3.8 shows the best candidate for leak rate, spectral radius, ridge, and input scaling among the search space for reservoir size of 500.

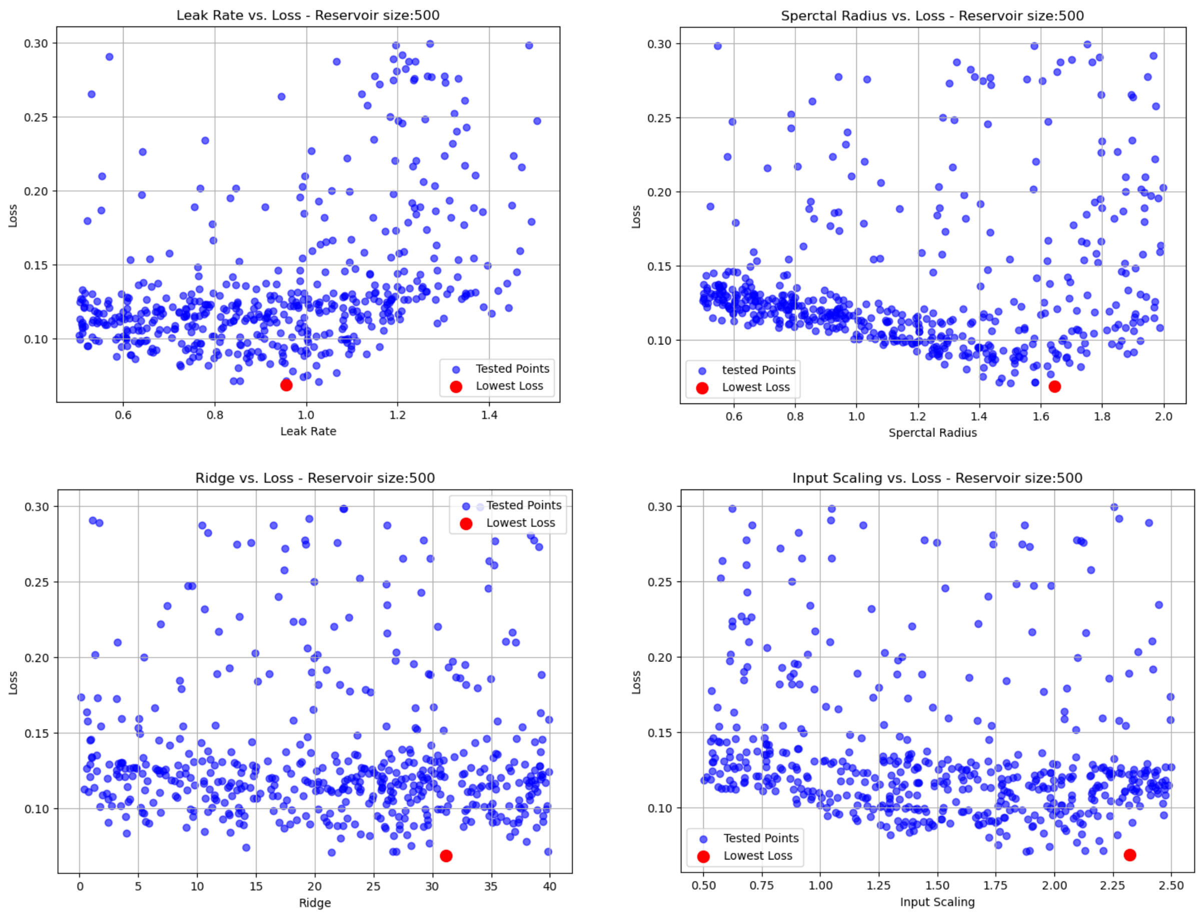

Figure 3.8 examined values (in blue) and best candidates (in red) for leaking rate, spectral radius, ridge, and input scaling

For plotting the results only solutions with MSE less than 0.3 have been considered.

3.3 Existing Temporal Convolutional Network

3.3.1 Architecture Design

The existing model has developed by (G. Subbicini, 2023) uses a bidirectional TCN (which infers based on both past and future tuples) with an input window of 5 seconds (15 tuples). The model receives a window of 15 samples from sensors and predicts the position corresponding to the 8th sample. The high-level architecture of the model has been depicted in Figure 3.9.

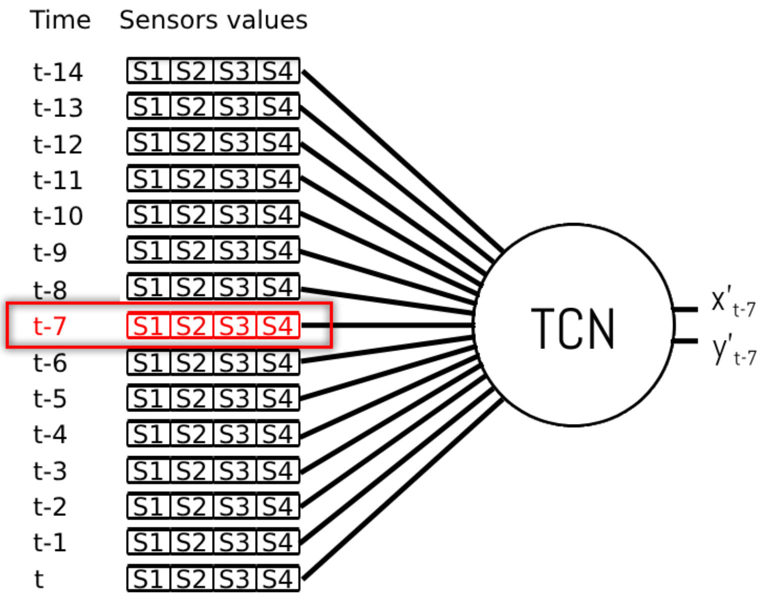

Figure 3.9 High level architecture of the TCN model. It receives a sequence of 15 samples of 4 sensors (4 channels). The output it x, y coordinates

3.3.2 Hyperparameter Tuning

For tuning the hyperparameter of the TCN Neural Architecture Search (NAS) was used. NAS seeks to automate the design of neural networks, aiming to achieve architectures that are equal to or better than manually designed ones. It focuses on optimizing the architecture of the network and predicting or testing its performance. NAS was implemented using AutoKeras (Zafar, 2022), an automated machine learning system built on Keras. AutoKeras employs a controller to generate neural network architectures based on a predefined grammar and encoding scheme. It also utilizes a searcher to evaluate these architectures based on criteria such as accuracy, complexity, and resource consumption. Finally, a trainer is used to train and validate the generated architectures. All relevant TCN parameters were optimized by NAS. The NAS was repeated 3 times for each NN and the AutoKeras tried 50 different combinations. For each combination it retrained 20 times for 800 epochs using Adamax optimizer tuned by AutoKeras. The best configuration can be seen in Table 3.3. With the mentioned hyperparameters setting the test loss of the TCN model is 0.065 m^2.

Table 3.3 Tuned parameters for TCN

Nb_filters	Dense	Kernel_size	Nb_layers	Nb_stacks	Dilation base

8	8	5	3	1	2

3.3.3 Training and Evaluation Procedures

The dataset was split into 3 parts, 60% of the dataset was allocated for training, 20% for cross-validation, and 20% for testing. The TCN receives a window of 15 samples. Each sample is composed of 4 sensor values at time t.

For the training purpose Adamax optimizer was used, and for the loss function mean square error (MSE) was selected. The metrics that were used for evaluation were MSE, and Average Distance Error (ADE).

The TCN optimized by NAS has inference accuracy (MSE = 0.065 m^2 , ADE = 0.309 m) and it uses 2034 resources (parameters to be learned).

3.4 Integration of Echo State Network and Temporal Convolutional Network

3.4.1 Proposed Hybrid Architecture

In the hybrid architecture I used the trained TCN network. Then I used the predicted X, Y values from the TCN as the input of the ESN model. For training the ESN model, the predictions of the TCN model are considered as the input values and ground truth X, Y values are considered as target values. The proposed architecture can be seen in Figure 3.9.

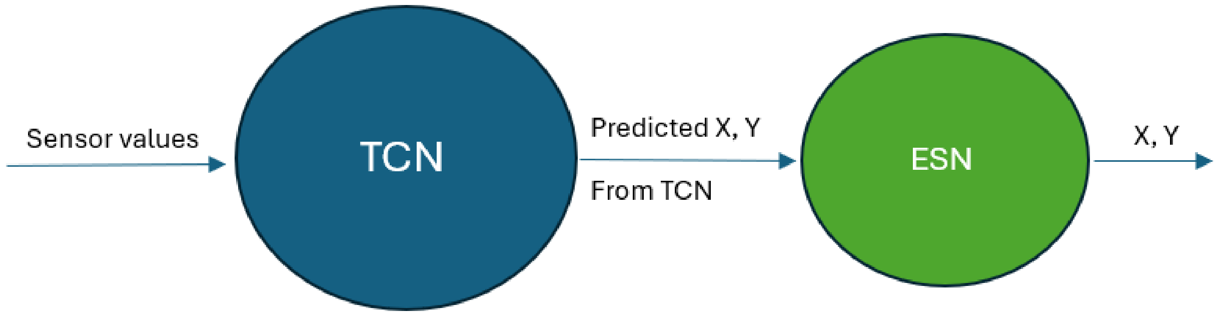

Figure 3.10 Architecture of the Integrated model

3.4.2 Training Strategy for the Integrated Model

The purpose of the integration of TCN and ESN is to increase the accuracy of the TCN with without increasing the complexity of the model too much. I keep the TCN model as it was. To train the ESN; which will be connected to the output of the TCN model; first, I created a dataset from the output of the TCN model. The structure of the dataset is as follows: the first two columns are the predicted x, y by TCN model, and the 3rd and 4th columns are the ground truth x, y.

For tuning the hyperparameters I proceeded as before using Hyperopt library. Different reservoir sizes were examined: 50, 100, 200, 400, 800, 1600 and for each reservoir size the following steps have been performed:

For each hyperparameter an interval was defined as follow:

- Spectral Radius: [0.1, 1.5]
- Leak Rate: [1e-2, 1.5]
- Input Scaling: [0.1, 2.5]
- Ridge: [1e-7, 1e2]

Then a random search was performed. 1000 points were selected randomly from search space. For each selection 100 different ESN were initialized and only points with loss < 0.3 were collected for plotting purposes.

The whole process repeated several times with modified intervals (shrunk intervals around the best-found solution). The comparison between test loss and reservoir size has been shown in Figure 3.10. The best reservoir size was 200 with test loss of 0.0565 m^2.

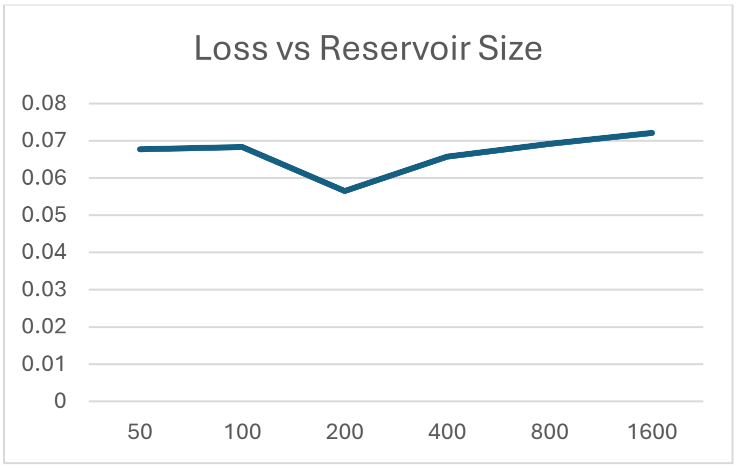

Figure 3.11 Loss vs Reservoir size for integrated model

The tested values for hyperparameters and the best candidates have been depicted in Figure 3.11. The best hyperparameters for each tested reservoir size with the corresponding test loss have been reported in Table 3.4.

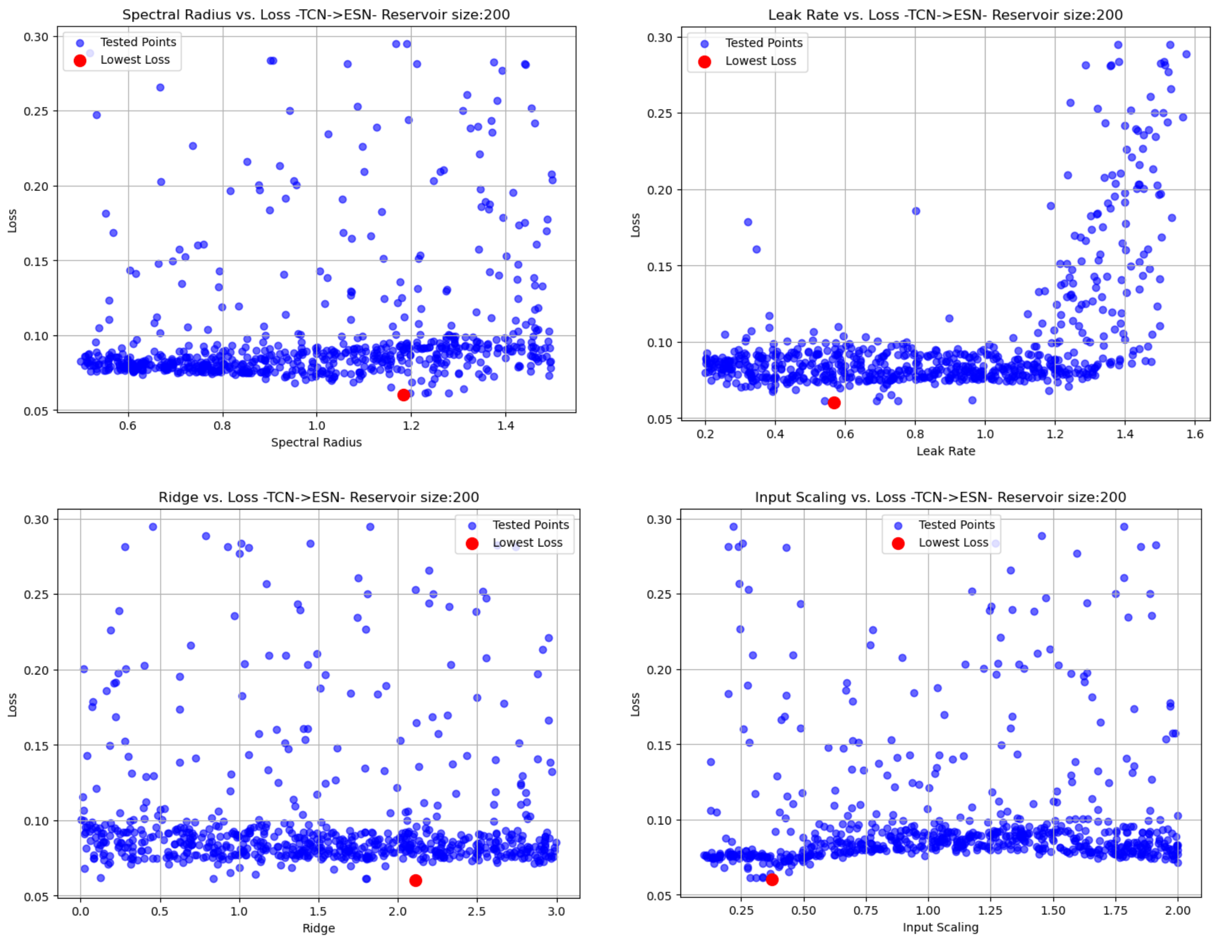

Figure 3.12 examined values (in blue) and best candidates (in red) for leaking rate, spectral radius, ridge, and input scaling for integrated model

Table 3.4 Best hyperparameters for each reservoir size for ESN in integrated model

Reservoir size	Leak rate	Spectral radius	Scaling	Ridge	Test loss (m^2)
50	0.5455	0.8476	0.2907	8.4615	0.0677
100	0.7247	0.9005	0.3010	30.1166	0.0683
200	**1.0468**	**1.2661**	**0.3926**	**0.1234**	**0.0565**
400	1.0763	0.9724	0.1074	20.792	0.0657
800	0.595	0.8352	1.857	12.7525	0.0692
1600	0.3866	1.258	0.2934	55.02	0.0721

3.5 Performance Metrics and Evaluation Criteria

In my study, I evaluated three models (Echo State Network (ESN), Temporal Convolutional Network (TCN), and a hybrid TCN->ESN model) by comparing their performance across several key metrics. I assessed accuracy using mean square error (MSE) and average Euclidean distance error (ADE), examined resource consumption by measuring the number of parameters in each model, and evaluated the smoothness based on Spectral Arc Length (SPARC) which is a smoothness metric often used in time-series analysis to assess the smoothness of trajectories or signals. The goal was to identify trade-offs between accuracy and computational efficiency, and to determine which model offers the best balance for indoor localization tasks using data from capacitive sensors. The result has been reported in the next chapter (Results and Analysis).

4. Results and Analysis

4.1 Echo State Network Performance in Indoor Localization

The Echo State Network (ESN) was implemented for indoor localization by predicting the position based on the input from four capacitive sensors. The performance was evaluated using the test set.

4.1.1 Accuracy and Precision

The ESN model was optimized using Hyperopt to fine-tune its hyperparameters, including reservoir size, spectral radius, leak rate, and input scaling. The mean square error (MSE) and average Euclidean distance error (ADE) were the performance metrics used. The best configuration achieved an MSE of 0.0614 m^2 and average Euclidean distance error (ADE) of 0.312 m with a reservoir size of 500.

In terms of smoothness the SPRAC on test set for ESN was 23.1 which indicates that ESN is a little bit smoother than TCN.

4.1.2 Computational Efficiency

The ESN's computational efficiency was evaluated by comparing resource consumption, specifically the number of parameters used. Due to its architecture, the ESN required fewer resources compared to traditional RNNs. The optimal model had a moderate computational overhead (1002) due to its larger reservoir size but maintained a good balance between accuracy and resource use.

4.2 Performance of the Existing Temporal Convolutional Network in Indoor Localization

The Temporal Convolutional Network (TCN) was also evaluated for the indoor localization task, trained on the same sensor data as the ESN.

4.2.1 Accuracy and Precision

Also, for TCN model the mean square error (MSE) and average Euclidean distance error (ADE) were the performance metrics used, the TCN achieved an MSE of 0.065 m^2 and an ADE of 0.309 meters.

In terms of smoothness the SPRAC on test set for TCN was 25.09 which is a little higher than that for ENS model (23.1).

4.2.2 Computational Efficiency

In terms of computational efficiency, the TCN required more resources than the ESN, with approximately 2034 parameters to be learned. Despite the larger resource consumption, the accuracy is almost the same for both modes.

4.3 Comparative Analysis of Echo State Network and Temporal Convolutional Network

Comparing the ESN and TCN, the ESN is more computationally efficient, requiring only 1002 parameters compared to the TCN's 2034. Also, in terms of smoothness it is a little bit smoother than TCN with SPARC of 23.1 for ESN and 25.09 for TCN.

Both models performed similarly in terms of ADE and MSE, with the ESN having a slight edge in MSE (0.0614 m^2), while the TCN had a marginally better ADE (0.309 m).

In summary, however, ESN in not much better than TCN in terms of accuracy and smoothness but it is by far more computationally efficient.

4.4 Performance of the Integrated Echo State Network-Temporal Convolutional Network Model

An integrated model combining the strengths of the ESN and TCN was developed to further enhance localization performance.

4.4.1 Improvement in Localization Accuracy

The integrated model achieved the lowest MSE of 0.0565 m^2 and the best ADE of 0.296 m. Its SPARC on Prediction (19.39) was significantly lower than both the ESN and TCN, providing the smoothest predictions.

4.4.2 Computational Overhead

Despite its improved performance, the integrated model required the most parameters (2406), which increased its computational cost. However, the significant improvements in both smoothness and accuracy justified the additional resources.

4.4.3 Generalization Capabilities

The integrated model showed the best generalization across different datasets, consistently achieving the smoothest predictions with the lowest SPARC on Prediction (19.39). It also delivered the best accuracy in terms of MSE and ADE, making it the most robust solution for indoor localization. This revised version reflects the better performance.

The complete comparison between three models; ESN, Integrated, and TCN(the baseline model); have been summarized in Table 5.1.

Table 5.1 comparison of ESN, Integrated model, and TCN (**baseline**)

Model	SPARC on Pred. (Test)	SPARC on GT (Test)	ADE (m)	MSE Train (m^2)	MSE Test (m^2)	parameters
TCN (baseline)	25.09	13.91	0.309	0.1342	0.065	2034
ESN	23.1	13.91	0.312	0.1069	0.0614	**1002**
Integrated	**19.39**	13.91	**0.296**	**0.0586**	**0.0565**	2406

The ESN remains the most resource-efficient model with 1002 parameters, followed by the TCN (2034), and the Integrated model (2406).

In terms of smoothness, The Integrated model offers the smoothest predictions with a SPARC on Prediction of 19.39, followed by the ESN (23.1) and the TCN (25.09).

The Integrated model offers the best balance between smoothness and accuracy, albeit at a higher computational cost, while the ESN is the most efficient in terms of resources but at a slight cost to smoothness and stability.

5. Discussion

5.1 Interpretation of Results

The three models—TCN, ESN, and the Integrated TCN-ESN—were compared based on smoothness (SPARC), accuracy (MSE, ADE), resource consumption (number of parameters).

The ESN showed the best computational efficiency, requiring the fewest parameters (1002) and maintaining good accuracy with a low MSE of 0.0614 m^2 and an ADE of 0.312 m. However, it produced less smooth predictions compared with the integrated model.

The TCN delivered almost the same smoothness as the ESN, with a SPARC on Prediction of 25.09, and maintained a slightly better ADE (0.309 m) and almost the same MSE (0.065 m^2). However, the computational cost of the TCN is by far higher than ESN. (2034 parameters for TCN and 1002 parameters for ESN).

The Integrated model, achieving the best accuracy, with an MSE of 0.0565 m^2 and ADE of 0.296 m. It also had the smoothest predictions, with a SPARC on Prediction of 19.39, but required the most parameters (2406).

In summary, the Integrated model strikes the best balance between accuracy and smoothness, while the ESN remains the most resource-efficient solution.

5.2 Strengths and Limitations of the Proposed Approach

The hybrid (TCN-ESN) model demonstrates a clear improvement in localization accuracy compared to standalone models, offering the lowest MSE and ADE.

In terms of smoothness, the integrated model exhibits the smoothest trajectory predictions (low SPARC), which is crucial for applications where motion smoothness matters.

In terms of resource consumption, the ESN's ability to maintain good accuracy with minimal resource consumption makes it ideal for real-time applications in resource-constrained environments.

However, the proposed model also has some limitations. One limitation is the computational overhead in integrated model. The hybrid model, while improving performance, requires significantly more parameters (2406), making it less suitable for applications with strict resource constraints.

The other limitation is the parameter sensitivity. All models are sensitive to hyperparameter tuning, especially the ESN, which requires careful adjustment of reservoir size and other hyperparameters.

5.3 Comparison with Existing Indoor Localization Methods

Compared to other indoor localization techniques like Wi-Fi-based and RFID systems, the ESN and TCN models proved to be more adaptive to capacitive sensor data, offering greater accuracy without the need for active signals or wearable tags. The hybrid model, in particular, surpassed traditional methods in terms of precision while also maintaining lower resource consumption, making it a strong candidate for real-time applications.

5.4 Practical Implications and Potential Applications

The ESN offers a lightweight solution for real-time indoor localization in environments where computational resources are limited, such as battery-powered devices, IoT systems, and low-power microcontrollers.

The Integrated TCN-ESN model, while more resource-intensive, is suited for applications where high accuracy and smoothness are critical, such as robotic navigation, augmented reality (AR), and autonomous vehicles.

Potential Applications:

The high accuracy and smooth trajectory predictions of the Integrated model make it ideal for drones, autonomous robots, and vehicles navigating complex indoor spaces.

The ESN’s efficiency makes it suitable for wearable devices tracking patient movements in real-time, where smoothness is less critical, but resource efficiency is paramount.

Applications requiring smooth and accurate position tracking, such as AR systems for indoor navigation in malls, airports, or museums, can benefit from the Integrated TCN-ESN model's performance.

The ESN's low resource consumption makes it a strong candidate for smart home systems that rely on real-time user localization for automation and safety features

6. Conclusion

6.1 Summary of Key Findings

This study investigated the performance of three models (TCN, ESN, and an Integrated TCN-ESN model) on an indoor localization task using data from capacitive sensors. The models were evaluated on various metrics, including accuracy (MSE and ADE), smoothness (SPARC), and resource efficiency (number of parameters).

The ESN model demonstrated the best computational efficiency, requiring the least number of parameters (1002) while maintaining a good balance between accuracy (MSE: 0.0614 m^2, ADE: 0.312 m).

The ESN offered slightly smoother predictions (SPARC on Prediction: 23.1) than TCN with almost the same accuracy but at a lower computational cost (1002 parameters).

The Integrated TCN-ESN model delivered the best overall performance, with the lowest MSE (0.0565 m^2), ADE (0.296 m), and SPARC on Prediction (19.39). However, this improvement came at the cost of higher resource usage (2406 parameters).

The key trade-off observed was between accuracy and resource consumption, with the Integrated model providing the best accuracy and smoothness, while the ESN was the most resource efficient.

6.2 Contributions to the Field of Indoor Localization

The study highlighted the potential of Echo State Networks in indoor localization tasks, showing that they can achieve competitive accuracy with significantly fewer computational resources.

The Integrated TCN-ESN model was shown to improve localization accuracy and smoothness, combining the strengths of both models. This hybrid approach sets a new standard for balancing accuracy and resource consumption in dynamic indoor environments.

The research identified key trade-offs between model smoothness, accuracy, and resource consumption, which can guide future system design for real-time localization tasks.

6.3 Future Research Directions and Recommendations

Model Optimization: Future research could focus on optimizing the Integrated TCN-ESN model to reduce the number of parameters while maintaining high accuracy and smoothness. Techniques like pruning and quantization could be explored to make the model more efficient for deployment in resource-constrained environments.

Exploring Other Hybrid Architectures: Further investigation into other hybrid architectures, such as combining ESNs with LSTMs or transformers, could provide additional gains in performance while improving generalization to more complex environments. Also, more advanced architectures of ESN such as using feedback loops, cascaded reservoirs and readouts, and using separated reservoirs for predicting x and y separately can be investigated.

Multimodal Sensor Fusion: To improve robustness, future work could integrate multimodal sensor data, such as, Wi-Fi signals, and vision-based inputs, along with capacitive sensors, to enhance localization accuracy across a broader range of environments and conditions.

References

[1] Ahson, S. A. (2010). *Location-based services handbook: Applications, technologies, .* Boca Raton: CRC Press.

[2] Alec Radford, L. M. (2016). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. Retrieved from https://ar5iv.labs.arxiv.org/html/1511.06434

[3] Alkhatib, A. (2011). A Review of Wireless Sensor Networks Applications. *The 2011 Conference on Innovations in Computing and Engineering Machinery*, (pp. 3-8).

[4] Bordoy, J. a. (2015). Single transceiver device-free indoor localization using ultrasound body reflections and walls., (pp. 1-7).

[5] G. Subbicini, L. L. (2023). Enhanced Exploration of Neural Network Models for Indoor Human Monitoring. *2023 9th International Workshop on Advances in Sensors and Interfaces (IWASI)*.

[6] Gallicchio, C. (2011). *Reservoir Computing for Learning in Structured Domains.* Retrieved from https://www.researchgate.net/publication/313238622_Reservoir_Computing_for_Learning_in_Structured_Domains

[7] Herbert Jaeger, H. H. (2004). *Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication.* Science.

[8] Jaeger, H. (2001). The" echo state" approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*.

[9] Lea, C. a. (2017). Temporal Convolutional Networks for Action Segmentation and Detection. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 1003-1012). IEEE.

[10] M. Kuki, H. N. (2013). Multi-human locating in real environment by thermal sensor. *IEEE International Conference on Systems, Man, and Cybernetics.* Manchester: IEEE.

[11] Nirjon, S. a. (2014). COIN-GPS: indoor localization from direct GPS receiving. *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, (pp. 301-314). Bretton Woods, New Hampshire, USA.

[12] Obeidat, H. a.-A. (2021). A Review of Indoor Localization Techniques and Wireless Technologies. *Wireless Personal Communications*.

[13] Obeidat, H. S. (2021). A Review of Indoor Localization Techniques and Wireless Technologies. *Wireless Pers Commun 119*, 289–327.

[14] Ramezani Akhmareh, A. M. (2016). A tagless indoor localization system based on capacitive sensing technology. *Sensors*.

[15] Raphael Wimmer, M. K. (2007). A capacitive sensing toolkit for pervasive activity detection and recognition. *Fifth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom'07)* (pp. 171-180). IEEE.

[16] Shaojie Bai, J. Z. (2018). An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling. *ArXiv*.

[17] T. N. Sainath, O. V. (2015). Convolutional, Long Short-Term Memory, fully connected Deep Neural Networks. *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 4580-4584). Brisbane, QLD, Australia,: IEEE.

[18] Tobias Grosse-Puppendahl, C. H. (2017). Finding common ground: A survey of capacitive sensing in human-computer. *Proceedings of the 2017 CHI Conference on Human Factors* (pp. 3293–3315). ACM.

[19] W. Maass, T. N. (2002). Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural Computation*.

[20] Xu, R. a. (2015). A New Indoor Positioning System Architecture Using GPS Signals. *Sensors*, 10074-10087.

[21] Y. Kim, S. H. (2015). Human detection using Doppler radar based on physical characteristics of targets. *IEEE Geoscience and Remote Sensing Letters, vol 12*, 289 – 293.

[22] Zafar, A. a. (2022). A Comparison of Pooling Methods for Convolutional Neural Networks. *Applied Sciences*.

Appendix A. Python Codes

Echo state network model (4 inputs, 2 outputs)

```python
from reservoirpy.nodes import Reservoir, Ridge
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
import reservoirpy as rpy
rpy.verbosity(0) # no need to be too verbose here
from reservoirpy.observables import nrmse, rsquare

# Load the dataset
train_data = pd.read_csv('train.csv')
test_data = pd.read_csv('test.csv')
# Split into inputs (first 4 columns) and output X, y (5th, 6th column)
X_train = train_data.iloc[:, :4].values
Y_train = train_data.iloc[:, 4:6].values
X_test = test_data.iloc[:, :4].values
Y_test = test_data.iloc[:, 4:6].values
dataset = (X_train, Y_train, X_test, Y_test)

# Objective functions accepted by ReservoirPy must respect some conventions:
#  - dataset and config arguments are mandatory, like the empty '*' expression.
#  - all parameters that will be used during the search must be placed after the *.
#  - the function must return a dict with at least a 'loss' key containing the result
# of the loss function. You can add any additional metrics or information with other
# keys in the dict. See hyperopt documentation for more informations.
def objective(dataset, config, *, input_scaling, N, sr, lr, ridge, seed):
    # This step may vary depending on what you put inside 'dataset'
    x_train, y_train, x_test, y_test = dataset

    # You can access anything you put in the config
    # file from the 'config' parameter.
    instances = config["instances_per_trial"]

    # The seed should be changed across the instances,
    # to be sure there is no bias in the results
    # due to initialization.
    variable_seed = seed

    losses = []; r2s = [];losses2 = [];
    for n in range(instances):
        # Build your model given the input parameters
        reservoir = Reservoir(
            units=N,
            sr=sr,
            lr=lr,
            input_scaling=input_scaling,
            seed=variable_seed
        )

        readout = Ridge(ridge=ridge)

        model = reservoir >> readout


        # Train your model and test your model.
        predictions = model.fit(x_train, y_train, warmup=10) \
                           .run(x_test)

        loss = nrmse(y_test, predictions, norm_value=np.ptp(x_train))
        loss2 = mean_squared_error(y_test, predictions)
        r2 = rsquare(y_test, predictions)

        # Change the seed between instances
        variable_seed += 1

        losses.append(loss)
        losses2.append(loss2)
        r2s.append(r2)

    # Return a dictionnary of metrics. The 'loss' key is mandatory when
    # using hyperopt.
    return {'loss': np.mean(losses2),
            'loss min': np.min(losses2),
            'r2': np.mean(r2s),
            'lr':lr,
            'sr':sr,
            'scaling':input_scaling,
            'seed':variable_seed}
    #return {'loss': np.mean(losses),
    #          'MSE': np.mean(losses2)}

import json
import random

# Generate random seeds
random_seed = random.randint(0, 1000000)          # Global random seed
esn_random_seed = random.randint(0, 1000000)
N = 500
hyperopt_config = {
    "exp": f"hyperopt-multiscroll-4-2-train-{N}-tpe",    # the experimentation name
    "hp_max_evals": 1000,              # the number of differents sets of parameters hyperopt has to try
    "hp_method": "tpe",                # the method used by hyperopt to chose those sets (see below)
    "seed": random_seed,               # the random state seed, to ensure reproducibility
    "instances_per_trial": 100,          # how many random ESN will be tried with each sets of parameters
    "hp_space": {                      # what are the ranges of parameters explored
        "N": ["choice",N],             # the number of neurons is fixed to 500
        "sr": ["uniform", 0.8, 1.9],   # the spectral radius is log-uniformly distributed between 1e-2 and 10
        #"sr": ["choice", 0.3, 0.6, 0.9, 1.2],
        #"sr": ["choice", 1.524246587049925],
        "lr": ["uniform", 0.5, 1.2],    # idem with the leaking rate, from 1e-3 to 1
        #"lr": ["choice", 0.8972398556015695],
        #"lr": ["choice", 0.8],
        #"input_scaling": ["choice", 2,1,0.5], # the input scaling is fixed
        "input_scaling": ["uniform", 0.8, 2.1],
        "ridge": ["uniform", 1e-1, 40],        # and so is the regularization parameter.
        #"ridge": ["choice", 1e-4, 1e-3, 1e-2, 1e-1, 1e-0, 1e1, 20, 25, 30],
        #"ridge": ["choice", 27.937098691397335],        # and so is the regularization parameter.
        "seed": ["choice", esn_random_seed]          # an other random seed for the ESN initialization
    }
}

# we precautionously save the configuration in a JSON file
# each file will begin with a number corresponding to the current experimentation run number.
with open(f"{hyperopt_config['exp']}.config.json", "w+") as f:
    json.dump(hyperopt_config, f)

from reservoirpy.hyper import research
best = research(objective, dataset, f"{hyperopt_config['exp']}.config.json", ".")
```

TCN model

```python
import numpy as np
import json
import csv
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import math
import os
import pathlib
import pandas as pd
import sys
import argparse
import tensorflow as tf
import tcn

from keras import regularizers, optimizers, losses, initializers, metrics
from sklearn.metrics import mean_squared_error
from keras.callbacks import ModelCheckpoint, EarlyStopping
from keras.models import load_model, model_from_json
from keras import layers, models
from keras import backend as K
from keras.utils import to_categorical
from tensorflow.keras.preprocessing import timeseries_dataset_from_array
from tcn import TCN, tcn_full_summary, compiled_tcn

#TCN model
def TCNmodel(nb_filters,num_units,kernel_size,hidden,nb_stacks,input_shape):
    x = layers.Input(shape=input_shape)
    tcn_out = TCN(nb_filters=nb_filters, kernel_size=kernel_size, nb_stacks=1, dilations
    =[2 ** i for i in range(hidden)], padding = 'same', use_skip_connections='True',
    dropout_rate=0.01, return_sequences=False ,activation='relu', kernel_initializer=
    'glorot_uniform', use_layer_norm= True, name='tcn')(x)
    flatten_out = layers.Flatten()(tcn_out)
    dense_out1 = layers.Dense(num_units,activation='linear',kernel_initializer=
    initializers.glorot_uniform())(flatten_out)
    dense_out = layers.Dense(2)(dense_out1)

    saved_model = models.Model(x, dense_out)

    return saved_model
#Load Dataset
def load_dataset():
    training_data = "preprocessed-training-Osama.csv"
    validation_data = "preprocessed-validation-Osama.csv"
    testing_data = "preprocessed-testing-Osama.csv"
    trainList = list()
    valList = list()
    testList = list()

    with open(training_data, 'r') as train_inp_csv:
        train_inp_csv_reader = csv.reader(train_inp_csv)
        for rowTr in train_inp_csv_reader:
            trainList.append(rowTr)

    trainArray = np.asarray(trainList, dtype=np.float32)
    #print(trainArray.shape)
    train_X = trainArray[:, 0:4]
    train_Y = trainArray[:, 4:6]
    #####################################

    with open(validation_data, 'r') as val_inp_csv:
        val_inp_csv_reader = csv.reader(val_inp_csv)
        for rowVl in val_inp_csv_reader:
            valList.append(rowVl)

    valArray = np.asarray(valList, dtype=np.float32)
    #print(valArray.shape)
    val_X = valArray[:, 0:4]
    val_Y = valArray[:, 4:6]
    #####################################

    with open(testing_data, 'r') as test_inp_csv:
        test_inp_csv_reader = csv.reader(test_inp_csv)
        for rowTe in test_inp_csv_reader:
            testList.append(rowTe)

    testArray = np.asarray(testList, dtype=np.float32)
    #print(valArray.shape)
    test_X = testArray[:, 0:4]
    test_Y = testArray[:, 4:6]



    X_train = np.array(train_X)    #np.transpose(train_X)
    Y_train = np.array(train_Y)    #np.transpose(train_Y)


    X_val = np.array(val_X)    #np.transpose(test1_X)
    Y_val = np.array(val_Y)    #np.transpose(test1_Y)


    X_test = np.array(test_X)    #np.transpose(test2_X)
    Y_test = np.array(test_Y)    #np.transpose(test2_Y)


    plt.plot(Y_train[:,0],Y_train[:,1],'b')
    plt.plot(Y_val[:,0],Y_val[:,1],'b')
    plt.plot(Y_test[:,0],Y_test[:,1],'b')
    plt.legend(['training','validation',"test"])

    return X_train, Y_train, X_val, Y_val, X_test, Y_test

#TCN Dataset
#Y--> (X,Y) coordinates of the path
def create_tf_dataset_tcn(window_size, X_train, Y_train, X_val, Y_val, X_test, Y_test):

    len_train = Y_train.shape[0]
    len_val = Y_val.shape[0]
    len_test = Y_test.shape[0]

    window_size = window_size
    centre_of_window = math.floor(window_size/2)

    sliced_train_X = np.zeros((len_train - window_size, window_size, 4))
    sliced_train_Y = np.zeros((len_train - window_size, 2))

    for index in range (window_size, len_train):
        sliced_train_X[index - window_size, :,:] = X_train[index - window_size : index,
        :]
        sliced_train_Y[index - window_size,:] = Y_train[index - centre_of_window, :]
    ###############################################################
    sliced_val_X = np.zeros((len_val - window_size, window_size, 4))
    sliced_val_Y = np.zeros((len_val - window_size, 2))

    for ind_val in range (window_size, len_val):
        sliced_val_X[ind_val - window_size, :,:] = X_val[ind_val - window_size : ind_val,
         :]
        sliced_val_Y[ind_val - window_size,:] = Y_val[ind_val - centre_of_window, :]
    ###############################################################
    sliced_test_X = np.zeros((len_test - window_size, window_size, 4))
    sliced_test_Y = np.zeros((len_test - window_size, 2))

    for ind_test in range (window_size, len_test):
        sliced_test_X[ind_test - window_size, :,:] = X_test[ind_test - window_size :
```

```
        ind_test, :]
        sliced_test_Y[ind_test - window_size,:] = Y_test[ind_test - centre_of_window, :]
    ###############################################################


    return sliced_train_X, sliced_train_Y, sliced_val_X, sliced_val_Y, sliced_test_X,
    sliced_test_Y


parent_dir =  "/home/Giorgia2/TCNBest_eval/"
window_time = 5
window_size = 15

nb_filters=16
dense=8
kernel_size=5
hidden=3
nb_stacks=1

input_sequence_length = 15
output_sequence_length = 1
directory = 'TCNautokeras'
path = os.path.join(parent_dir, directory)
if not os.path.exists(path):
    os.mkdir(path)
    print("Directory '%s' created" %directory)

X_train, Y_train, X_val, Y_val, X_test, Y_test= load_dataset()
X_train_tcn,Y_train_tcn,X_val_tcn, Y_val_tcn,X_test_tcn, Y_test_tcn =
create_tf_dataset_tcn(15,X_train, Y_train,X_val, Y_val,X_test, Y_test)

#Load model
saved_model = TCNmodel(nb_filters,dense,kernel_size,hidden,nb_stacks,X_train_tcn[1,:,:].
shape)

saved_model.load_weights(path+'/best_model.h5')
saved_model.compile('adam','mse')

trnable_params = saved_model.count_params()

eval_test = saved_model.evaluate(X_test_tcn, Y_test_tcn, batch_size = None , verbose=1)
print("test: ", eval_test)


eval_val = saved_model.evaluate(X_val_tcn, Y_val_tcn, batch_size = None , verbose=1)
print("validation: ", eval_val)


eval_train = saved_model.evaluate(X_train_tcn, Y_train_tcn, batch_size = None , verbose=1
)
print("training: ", eval_train)


preds_train_tcn = (saved_model.predict(X_train_tcn, batch_size = None))
preds_val_tcn = (saved_model.predict(X_val_tcn, batch_size = None))
preds_test_tcn = (saved_model.predict(X_test_tcn, batch_size = None))




plt.figure()
plt.plot(Y_train_tcn[:,0],Y_train_tcn[:,1])
plt.plot(preds_train_tcn[:,0],preds_train_tcn[:,1])
plt.legend(['true','pred'])
# plt.title('Square')
plt.xlim(0,2.5)
plt.ylim(0,2.5)
```

```python
plt.savefig(path +'/Training_pred.pdf')
plt.close()



plt.figure()
plt.plot(Y_val_tcn[:,0],Y_val_tcn[:,1])
plt.plot(preds_val_tcn[:,0],preds_val_tcn[:,1])
plt.legend(['true','pred'])
# plt.title('Diag')
plt.xlim(0,2.5)
plt.ylim(0,2.5)
plt.savefig(path +'/Validation_pred.pdf')
plt.close()

plt.figure()
plt.plot(Y_test_tcn[:,0],Y_test_tcn[:,1])
plt.plot(preds_test_tcn[:,0],preds_test_tcn[:,1])
plt.legend(['true','pred'])
# plt.title('Circle')
plt.xlim(0,2.5)
plt.ylim(0,2.5)
plt.savefig(path +'/Testing_pred.pdf')
plt.close()


mse_test=mean_squared_error(Y_test_tcn[:,:],preds_test_tcn[:,:])
mse_train=mean_squared_error(Y_train_tcn[:,:],preds_train_tcn[:,:])
mse_val=mean_squared_error(Y_val_tcn[:,:],preds_val_tcn[:,:])


dist_euc_train = np.mean(np.sqrt(np.sum(np.square(preds_train_tcn[:,:] - Y_train_tcn
[:,:]), axis=-1)))
dist_euc_dev = np.mean(np.sqrt(np.sum(np.square(preds_val_tcn[:,:] - Y_val_tcn[:,:]),
axis=-1)))
dist_euc_test = np.mean(np.sqrt(np.sum(np.square(preds_test_tcn[:,:] - Y_test_tcn[:,:]),
axis=-1)))

#first derivative
vel_train = np.zeros([Y_train_tcn.shape[0]-1, 2])
vel_pred_train = np.zeros([Y_train_tcn.shape[0]-1, 2])
for i in (1, Y_train_tcn.shape[0]):
    vel_train[i-1,:]= abs(Y_train_tcn[i,:]-Y_train_tcn[i-1,:])*3
    vel_pred_train[i-1,:]= abs(preds_train_tcn[i,:]-preds_train_tcn[i-1,:])*3

vel_val = np.zeros([Y_val_tcn.shape[0]-1, 2])
vel_pred_val = np.zeros([Y_val_tcn.shape[0]-1, 2])
for i in (1, Y_val_tcn.shape[0]):
    vel_val[i-1,:]= abs(Y_val_tcn[i,:]-Y_val_tcn[i-1,:])*3
    vel_pred_val[i-1,:]= abs(preds_val_tcn[i,:]-preds_val_tcn[i-1,:])*3

vel_test = np.zeros([Y_test_tcn.shape[0]-1, 2])
vel_pred_test = np.zeros([Y_test_tcn.shape[0]-1, 2])
for i in (1, Y_test_tcn.shape[0]):
    vel_test[i-1,:]= abs(Y_test_tcn[i,:]-Y_test_tcn[i-1,:])*3
    vel_pred_test[i-1,:]= abs(preds_test_tcn[i,:]-preds_test_tcn[i-1,:])*3

mse_vel_train = mean_squared_error(vel_train[:,:],vel_pred_train[:,:])
mse_vel_val = mean_squared_error(vel_val[:,:],vel_pred_val[:,:])
mse_vel_test = mean_squared_error(vel_test[:,:],vel_pred_test[:,:])

rmse_vel_train = math.sqrt(mse_vel_train)
rmse_vel_val = math.sqrt(mse_vel_val)
rmse_vel_test = math.sqrt(mse_vel_test)


#second derivative
```

```
acc_train = np.zeros([vel_train.shape[0]-1, 2])
acc_pred_train = np.zeros([vel_train.shape[0]-1, 2])
for i in (1, vel_train.shape[0]):
    acc_train[i-1,:]= abs(vel_train[i,:]-vel_train[i-1,:])*3
    acc_pred_train[i-1,:]= abs(vel_pred_train[i,:]-vel_pred_train[i-1,:])*3

acc_val = np.zeros([vel_val.shape[0]-1, 2])
acc_pred_val = np.zeros([vel_val.shape[0]-1, 2])
for i in (1, vel_val.shape[0]):
    acc_val[i-1,:]= abs(vel_val[i,:]-vel_val[i-1,:])*3
    acc_pred_val[i-1,:]= abs(vel_pred_val[i,:]-vel_pred_val[i-1,:])*3

acc_test = np.zeros([vel_test.shape[0]-1, 2])
acc_pred_test = np.zeros([vel_test.shape[0]-1, 2])
for i in (1, vel_test.shape[0]):
    acc_test[i-1,:]= abs(vel_test[i,:]-vel_test[i-1,:])*3
    acc_pred_test[i-1,:]= abs(vel_pred_test[i,:]-vel_pred_test[i-1,:])*3

mse_acc_train =mean_squared_error(acc_train[:,:],acc_pred_train[:,:])
mse_acc_val =  mean_squared_error(acc_val[:,:],acc_pred_val[:,:])
mse_acc_test = mean_squared_error(acc_test[:,:], acc_pred_test[:,:])

rmse_acc_train = math.sqrt(mse_acc_train)
rmse_acc_val = math.sqrt(mse_acc_val )
rmse_acc_test =math.sqrt(mse_acc_test)

#Write results:
title='/nn-results.csv'
results = open(path+title, "w+")

results.write('"Training MSE",')
results.write('"Validation MSE",')
results.write('"Testing MSE",')

results.write('"Training MSE",')
results.write('"Validation MSE",')
results.write('"Testing MSE",')

results.write('"Traingn Euclidean",')
results.write('"Validation Euclidean",')
results.write('"Testing Euclidean",')

results.write('"Traingng First derivative RMSE",')
results.write('"Validation First derivative RMSE",')
results.write('"Testing First derivative RMSE",')

results.write('"Traingng Second derivative RMSE",')
results.write('"Validation Second derivative RMSE",')
results.write('"Testing Second derivative RMSE"')

results.write('\n')

results.write('%f,' % eval_train)
results.write('%f,' % eval_val)
results.write('%f,' % eval_test)

results.write('%f,' % mse_train)
results.write('%f,' % mse_val)
results.write('%f,' % mse_test)

results.write('%f,' % dist_euc_train)
results.write('%f,' % dist_euc_dev)
results.write('%f,' % dist_euc_test)

results.write('%f,' % rmse_vel_train)
results.write('%f,' % rmse_vel_val)
results.write('%f,' % rmse_vel_test)

results.write('%f,' % rmse_acc_train)
results.write('%f,' % rmse_acc_val)
results.write('%f ' % rmse_acc_test)

results.write('\n')

results.close()

```

ESN 2 Inputs 2 outputs

```python
#Optimize hyperparameters
from reservoirpy.observables import nrmse, rsquare
from reservoirpy.nodes import Reservoir, Ridge
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
import reservoirpy as rpy
rpy.verbosity(0) # no need to be too verbose here
from hyperopt import hp

# define the objrctive function
# Objective functions accepted by ReservoirPy must respect some conventions:
#  - dataset and config arguments are mandatory, like the empty '*' expression.
#  - all parameters that will be used during the search must be placed after the *.
#  - the function must return a dict with at least a 'loss' key containing the result
# of the loss function. You can add any additional metrics or information with other
# keys in the dict. See hyperopt documentation for more informations.
def objective(dataset, config, *, input_scaling, N, sr, lr, ridge, seed):
    # This step may vary depending on what you put inside 'dataset'
    x_train, y_train, x_test, y_test = dataset

    # You can access anything you put in the config
    # file from the 'config' parameter.
    instances = config["instances_per_trial"]

    # The seed should be changed across the instances,
    # to be sure there is no bias in the results
    # due to initialization.
    variable_seed = seed

    losses = []; r2s = []; losses2=[]
    for n in range(instances):
        # Build your model given the input parameters
        reservoir = Reservoir(
            units=N,
            sr=sr,
            lr=lr,
            input_scaling=input_scaling,
            seed=variable_seed
        )

        readout = Ridge(ridge=ridge)

        model = reservoir >> readout


        # Train your model and test your model.
        predictions = model.fit(x_train, y_train, warmup=10) \
                           .run(x_test)

        loss = nrmse(y_test, predictions, norm_value=np.ptp(x_train))
        loss2 = mean_squared_error(y_test, predictions)
        r2 = rsquare(y_test, predictions)

        # Change the seed between instances
        variable_seed += 1

        losses.append(loss)
        losses2.append(loss2)
        r2s.append(r2)

    # Return a dictionnary of metrics. The 'loss' key is mandatory when
    # using hyperopt.
    return {'loss': np.mean(losses),
            'r2': np.mean(r2s),
            'mse': np.mean(losses2)}
```

```
import json
import random
# Generate random seeds
random_seed = random.randint(0, 1000000)           # Global random seed
esn_random_seed = random.randint(0, 1000000)
N = 200
hyperopt_config = {
    "exp": f"hyperopt-{N}-secondESN-multiscroll",    # the experimentation name
    "hp_max_evals": 200,               # the number of differents sets of parameters hyperopt has to try
    "hp_method": "tpe",            # the method used by hyperopt to chose those sets (see below)
    "seed": random_seed,                        # the random state seed, to ensure reproducibility
    "instances_per_trial": 10,          # how many random ESN will be tried with each sets of parameters
    "hp_space": {                      # what are the ranges of parameters explored
        "N": ["choice", N],              # the number of neurons is fixed to 500
        #"sr": ["uniform", 1e-1, 1.5],    # the spectral radius is log-uniformly distributed between 1e-2 and 10
        "sr": ["choice", 0.1, 0.3, 0.6, 0.9, 1.2],
        #"lr": ["uniform", 1e-1, 1.5],     # idem with the leaking rate, from 1e-3 to 1
        "lr": ["choice", 0.5, 0.7, 1, 1.2, 1.5],
        "input_scaling": ["choice", 1.0, 0.5, 2], # the input scaling is fixed

        "ridge": ["uniform", 1e-3, 30],         # and so is the regularization parameter.
        #"ridge": ["choice", 1e-8,1e-7,1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e-0, 1e1, 20, 30],
        "seed": ["choice", esn_random_seed]          # an other random seed for the ESN initialization
    }
}

# we precautionously save the configuration in a JSON file
# each file will begin with a number corresponding to the current experimentation run number.
with open(f"{hyperopt_config['exp']}.config.json", "w+") as f:
    json.dump(hyperopt_config, f)

# Prepare the data
import pandas as pd


# Load the dataset
train_data = pd.read_csv('pred_first_esn.csv', header=None)
# Split into inputs and output X
X_train = train_data.iloc[:, :].values
test_data = pd.read_csv('pred_first_esn_test.csv', header=None)
X_test = test_data.iloc[:, :].values


# Load the dataset
train_data = pd.read_csv('train2.csv')
test_data = pd.read_csv('test.csv')
Y_train = train_data.iloc[:, 4:6].values
Y_test = test_data.iloc[:, 4:6].values

dataset = (X_train, Y_train, X_test, Y_test)

from reservoirpy.hyper import research
best = research(objective, dataset, f"{hyperopt_config['exp']}.config.json", ".")

from reservoirpy.hyper import plot_hyperopt_report
fig = plot_hyperopt_report(hyperopt_config["exp"], ("lr", "sr", "ridge"), metric="r2")

plt.show()
```