TinyML-Empowered Indoor Positioning with Light

Model Optimization using Neural Architecture Search

Neel Lodha[1]

Supervisor(s): Qing Wang[1]**, Ran Zhu**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Neel Lodha
Final project course: CSE3000 Research Project
Thesis committee: Qing Wang, Ran Zhu, Ranga Rao Venkatesha Prasad

Abstract

Visible light positioning (VLP) systems are a promising solution for indoor positioning, utilizing light-emitting diodes (LEDs) as transmitters and photodiodes (PDs) as receivers. A received signal strength (RSS) based VLP system's accuracy is heavily dependent on the density of collected fingerprints, being a very labor-intensive process. In this study, we focus on RSS fingerprints to achieve centimetre level positioning accuracy, while addressing the challenges of labor-intensive fingerprint collection and deployment on resource-constrained devices like the Raspberry Pi Pico microcontroller. We found different neural network architectures using Neural Architecture Search (NAS) to optimize the VLP system, which achieve on average $12mm$ positioning error with low inference latency around $50ms$ on the Raspberry Pi Pico.

1 Introduction

Location-aware services are a crucial feature of modern wireless networks, enabling a wide range of essential applications, particularly those in the Internet of Things (IoT). Although wireless communication has advanced significantly in recent years, achieving accurate and cost-effective indoor positioning using wireless systems continues to be a major challenge. Numerous applications like smart retail, navigation in large public facilities (e.g., hospitals, malls), assisted living, and industrial tracking of robotic arms have driven exploration of wireless technologies such as WiFi [1], Bluetooth [2], Zigbee [3], and ultra-wideband (UWB) [4]. However, these technologies face persistent challenges, including limited accuracy, electromagnetic interference, and spectrum congestion [5].

The widespread adoption of light-emitting diodes (LEDs) for illumination, coupled with advances in visible light communication (VLC), has positioned visible light positioning (VLP) as a compelling alternative for indoor localization. VLP benefits from leveraging existing lighting infrastructure, offers high positioning accuracy, and provides enhanced security. Furthermore, LED-based systems operate in the unlicensed spectrum and are biologically safe, making them particularly attractive for environments such as hospital operating rooms, where radio frequency (RF) signals may pose risks to both patients and sensitive equipment.

Unlike RF-based systems, VLP is less susceptible to multipath reflection, resulting in improved positioning accuracy [6]. Several techniques have been proposed for VLP, including angle-of-arrival (AOA) [7], time-of-arrival (TOA), time-difference-of-arrival (TDOA) [8], and received signal strength (RSS) [9]. While AOA offers high accuracy, it requires complex computations and expensive hardware. TOA and TDOA demand highly sensitive receivers and precise synchronization between transmitters and receivers. In contrast, RSS-based methods are cost-effective and do not require synchronized infrastructure, making them suitable for resource-constrained environments.

VLP systems typically use multiple LEDs as transmitters and photodiodes (PDs) or cameras as receivers. Due to their lower cost and power consumption, PDs are preferred for scalable deployment. Prior work by Zhu et al. [10] addressed the labor-intensive nature of data collection in RSS-based VLP by introducing data preprocessing techniques like data cleaning and data augmentation. Building upon this foundation, our work focuses on optimizing the VLP model for deployment on microcontrollers. Specifically, we target improvements in positioning accuracy and inference latency through Tiny Machine Learning (TinyML), enabling efficient and scalable indoor localization on resource-constrained hardware.

Thus, this papers aims to answer the following research question and associate sub-questions:

Main Research Question

How can we improve the performance (accuracy, inference latency) of a VLP system running TinyML on resource-constrained devices?

RQ1	Which traditional neural network architectures (Convolutional Neural Networks, Multilayer Perceptrons) found using Neural Architecture Search are most suitable for our RSS based VLP system when deployed on a Raspberry Pi Pico microcontroller?
RQ2	How does the density of the fingerprint dataset impact the model's performance.

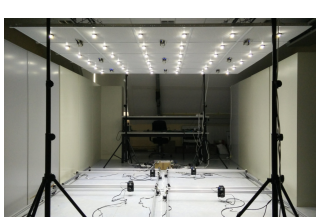
The remainder of this paper is structured as follows. Section 2 reviews prior work by Zhu et al. [10] and surveys relevant literature in RSS-based VLP, TinyML, and neural architecture search (NAS). Section 3 outlines our proposed methodology, including data preprocessing, NAS procedures, and optimization strategies. Section 4 details the experimental setup used to evaluate our approach. Section 5 presents the evaluation results, demonstrating improvements in positioning accuracy and reductions in inference latency. Section 6 situates our findings within the broader literature, highlighting limitations and directions for future research. Section 7 discusses the ethical considerations relevant to our study. Finally, Section 8 summarizes our research questions and key contributions.

2 Background

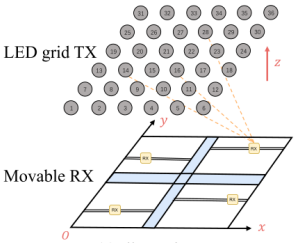
In this section, we discuss the state-of-the-art methods for RSS based VLP systems, the work done by Zhu et al. [10], related work in the field of TinyML and Neural Architecture Search (NAS).

2.1 RSS based VLP systems

VLP systems utilize the RSS from multiple light-emitting diodes (LEDs) to determine the position of a receiver. The RSS values are influenced by factors such as distance, angle, and environmental conditions. Previous research has shown that RSS-based VLP systems can achieve high accuracy in in-

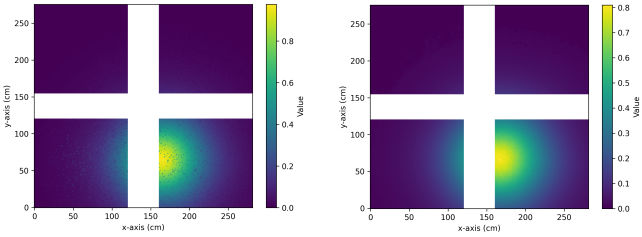


(a) Testbed



(b) Illustration

Figure 1: DenseVLC testbed, a) actual testbed, b) testbed illustration.



Raw data (1cm granularity) Clean data (1cm granularity)

Figure 2: Visualization of the LED 9 RSS values in the x-y plane.

door localization using various techniques like triangulation and fingerprinting [11].

Hsu et al. [12] proposed a Convolutional Neural Network (CNN) based VLP system along with data preprocessing techniques like cleaning and interpolation to improve accuracy. Similarly, Zhu et al. [10] achieved centimeter-level accuracy data preprocessing techniques like data cleaning and data augmentation and used machine learning models to reduce data collection while maintaining good position accuracy. This work serves as the foundation of our research, as we aim to further optimize the VLP system for deployment on resource-constrained devices using TinyML techniques.

2.2 Previous work by Zhu et al.

Dataset

Zhu et al. [10] used the DenseVLC testbed [13] (see Figure 1) to obtain the dataset for their VLP system.

The setup consists of 36 LED transmitters (TX) and 4 photodiode receivers (RX). The TXs are mounted on a height adjustable ceiling in a 6×6 grid with spacing of $0.5m$. The testbed is divided into 4 sections, with each section having an area of $1.2m \times 1.2m$. Each RX is assigned to a section and can move freely within along the x and y axes.

Every fingerprint sample in the dataset, consists of the received signal strength (RSS) values from all the 36 TXs, the RX's positional coordinates (x, y, z). Measurements are conducted at $1cm$ intervals in both the x and y direction. Similarly, measurements are taken at 2 different heights (vertical difference between the TX and RX) of $176cm$ and $192cm$. Finally, each measurement is taken 3 times at a specific location. In conclusion, the dataset consists of $351,384$ measurements.

Data preprocessing

Zhu et al. [10] proposed two data preprocessing to improve the performance of the model. The first one is a data cleaning setup which using the Lambertian model to remove the outliers (see Figure 2). The second one is using data augmentation where data is only collected at intervals of $8cm$ in the x and y direction. The data is then augmented by interpolating the data to $1cm$ intervals. This allows for a more dense dataset while keeping the number of measurements low. In our experiment, we will be using the same data cleaning and augmentation techniques to evaluate the performance of our model making sure that the results are comparable.

TinyML

To run the VLP system, Zhu et al. [10] implemented three different machine learning models. The first two are Random forests and Support Vector Machines (SVM) which are both non-deep learning models. The third is a deep learning model: a Multilayer Perceptron (MLP) comprising five hidden layers. To enable deployment on an Arduino Nano microcontroller, the MLP model was quantized, reducing its size from $5MB$ to $1.3MB$. 'This' MLP model will also serve as the baseline for our research.

2.3 Neural Architecture Search

Neural Architecture Search (NAS) is a method to automatically search for the best architecture for a given task. Figure 3 shows the general workflow of NAS. The search space is defined by the user, and the search algorithm explores this space to find the best architecture. The search process is usually expensive, as it requires training and evaluating many architectures. However, recent advances in NAS have made it possible to search for architectures more efficiently [14]. It has also been shown to be effective in finding architectures that outperform hand-designed architectures.

Recent research has increasingly focused on hardware-aware Neural Architecture Search (NAS), which aims to discover neural network architectures that are not only accurate but also optimized for deployment on specific hardware platforms, such as microcontrollers and specialized accelerators. For instance, uNAS [15] introduces a multi-objective constrained NAS algorithm specifically tailored for microcontrollers, balancing predictive performance with strict constraints on memory, storage, and inference latency, which are critical for resource-scarce devices. Similarly, Banbury et al. [16] utilizing NAS to deploy neural network architectures on commodity microcontrollers, while using a multi-objective optimization approach to balance accuracy, memory usage, and inference time.

In this research, we will be using NAS to find the best architecture for our VLP system. The search will be conducted under strict hardware constraints to ensure that the resulting model is compatible with deployment on a resource-limited microcontroller, specifically the Raspberry Pi Pico (Figure 4) featuring a dual-core ARM Cortex-M0+ processor running at up to $133MHz$, $264KB$ of SRAM, and $2MB$ of onboard flash memory, and we need to ensure that the model is small enough to fit on the microcontroller while still achieving good accuracy and inference latency.

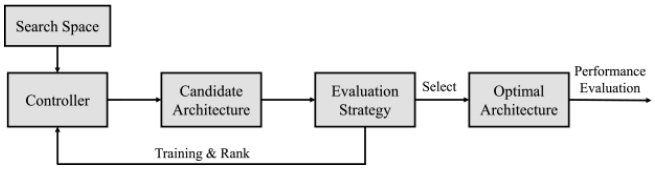


Figure 3: Neural Architecture Search workflow [14]

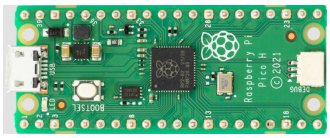


Figure 4: Raspberry Pi Pico H

2.4 Quantization

Quantization is a technique which performs computations and stores tensors at lower bit widths compare to floating point precision (e.g., 32-bit floating point). By reducing the bit width to for example 8-bit integers, the quantization enables more compact model representations and utilization of efficient vectorized operations on various hardware platforms [17, 18]. This technique is particularly beneficial during inference, significantly reducing computation costs while maintaining inference accuracy.

There are two methods for quantization: Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ). QAT involves training the model with quantization in mind, allowing the model to learn to adapt to the quantized weights and activations. PTQ, on the other hand, applies quantization after the model has been trained, which is often faster and requires less computational resources.

3 Methodology

This section introduces the methodology used to find different neural network architectures, outlines the different stages of the Neural Architecture Search (NAS) process, and describes the evaluation metrics used to assess the performance of the models.

3.1 Dataset

As mentioned in the previous section, we use the dataset from the DenseVLC testbed which contains 351,384 measurements. Each measurement consists of the RSS values from all 36 TXs (input values) and the RX's positional coordinates (x, y, z) (output values). A cleaned version of this dataset is prepared by applying the data cleaning technique proposed by Zhu et al. [10]. We can use both the original (raw) and the cleaned dataset for training the models and evaluating their performance. To keep things simple, we only use the measurements taken at the height of $176cm$, similar to the previous work [10]. To remove biases from the dataset, we ensure that both input and values are normalized using the min-max technique to a range of $[0, 1]$. Finally, the normalized dataset is split into training (70%), testing (20%) and validation (10%) sets.

A key distinction in our approach compared to previous work, such as Zhu et al. [10] which utilized Multi-Layer Perceptrons (MLPs) with a flattened array of 36 RSS values, is our input representation. Given that the 36 transmitters (LEDs) in the DenseVLC testbed are arranged in a physical grid, we reshape the input RSS values into a 6×6 matrix. This 2D representation could allow Convolutional Neural Network (CNN) models to potentially learn and exploit spatial features and correlations between neighbouring transmitters, which would be lost in a 1D flattened input. Thus, we also explore the use of CNNs in our search for optimal architectures.

3.2 Neural Architecture Search

We use the Microsoft Neural Network Intelligence (NNI) framework [19] to perform our Neural Architecture Search (NAS). NNI is a general open source AutoML toolkit for automating the process of hyperparameter tuning and neural architecture search. It provides a flexible and easy-to-use interface for defining search spaces, training models, and evaluating their performance. The toolkit supports state-state-of-the-art NAS algorithms, including both multi-shot and one-shot methods, and can be easily integrated with popular deep learning frameworks such as PyTorch and TensorFlow.

Search Space

Designing an effective search space is crucial for the success of a NAS process, especially when finding architectures for resource-constrained devices [20]. Zoph et al. [21] introduced a cell based search space where the search space is defined as a sequence of cells. A cell is a small sub-architecture that is repeated multiple times to form the final architecture. This approach allows to find the best cells that can be combined to form a larger architecture, while reducing the search space size and computational cost. Zoph et al. [21] introduced two kinds of cells: *normal cell* that preserves the input dimensions and a *reduction cell* that reduces the spatial dimensions.

For the CNN search space, we define it as a sequence of cells, similar to common NAS setups. We also test a simpler version with just a few convolutional layers, pooling and full connected (FC) layers, which lets us compare complex cell-based architectures with basic ones. For the MLP search space, the cell-based design doesn't apply, so we use a straightforward setup with only FC layers. Finally, the output layer is a fully connected layer with 2 neurons which are then activated using Sigmoid (activation function to ensure the output values are in the range of $[0, 1]$) corresponding to the (x, y) coordinates of the RX.

Search Algorithm

For the exploration of our defined search space, our methodology leverages the capabilities of the Microsoft NNI framework, which supports a variety of NAS search algorithms. These algorithms broadly include multi-shot techniques, exemplified by reinforcement learning (RL) based approaches [22], where an agent iteratively learns to select better architectures. Additionally, NNI accommodates more recent one-shot methods (see Figure 5) designed for greater computational efficiency. Prominent examples within this paradigm are:

• Efficient Neural Architecture Search (ENAS) [23]: This

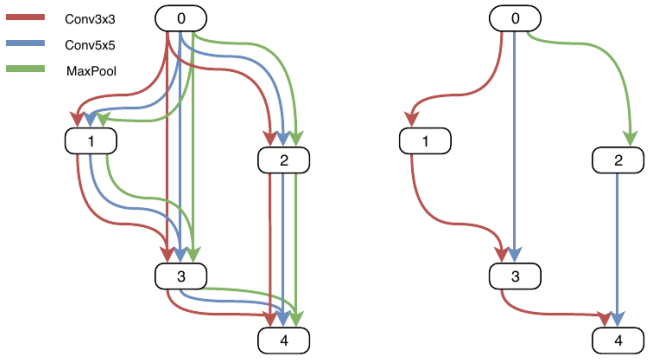


Figure 5: "Illustration of one-shot architecture search. Simple network with an input node (denoted as 0), three hidden nodes (denoted as 1,2,3) and one output node (de- noted as 4). Instead of applying a single operation (such as a 3x3 convolution) to a node, the one-shot model (left) contains several candidate operations for every node, namely 3x3 convolution (red edges), 5x5 convolution (blue edges) and MaxPooling (green edges) in the above illustration. Once the one-shot model is trained, its weights are shared across different architectures, which are simply subgraphs of the one-shot model (right)" [25, 24]

method employs a shared-parameter super-network and a reinforcement learning controller to guide the search process, allowing for efficient exploration of the architecture space.
• Differentiable Architecture Search (DARTS) [24]: This approach relaxes the architectural choices into a continuous domain, enabling gradient-based optimization, which significantly reduces the computational burden compared to traditional methods.
Both of these techniques are one-shot strategies where only a single super-net architecture is trained, contrary to the multi-shot methods like random search or grid search which require training each architecture in the search space independently, leading to significant computational overhead.

In our specific implementation, we will be utilizing both the ENAS and DARTS search strategies to explore the search space.

Crucially, to ensure the discovered architectures are suitable for our resource-constrained Raspberry Pi Pico target, we employ a custom evaluator. This evaluator performs a regression task, optimizing for Mean Squared Error (MSE) loss, which directly reflects the VLP system's objective of predicting (x,y) coordinates. To incorporate the stringent hardware limitations, we integrate hardware-aware penalties directly into this MSE loss function. Specifically, penalties are added based on:
• The number of model parameters: This directly correlates with the model size, ensuring it fits within the Pico's limited memory.
• The number of operations: This serves as a proxy for inference latency.

3.3 Deployment

After the NAS process is completed, the best architecture is exported as PyTorch models which are then converted to TFlite-micro format using the Tensorflow Lite Converter. Since TFlite only supports 8-bit quantized models, we have to quantize the default float 32-bit models to 8-bit. This means, that the model size is roughly reduced by a factor of 4.

Quantization

To quantize the models, we use the post-training quantization technique provided by the Tensorflow Lite Converter. 300 samples (forming the representative dataset) from the validation set are used to calibrate the quantization process. The input tensors and output tensors are not quantized, only the weights and the biases of the model are. Since the Pico only has $2MB$ of flash memory, the models need to be quantized to fit within this limit. Thus, we can set an upper limit of approximately $7MB$ for the non quantized model size, which would be reduced to approximately $1.75MB$ post quantization, allowing for the weights to fit alongside the pico-tflmicro and pico-sdk libraries in the flash. We can also set an upper limit of $800KB$ for the non quantized model size, which would be reduced to approximately $200KB$ post quantization, allowing for the entire model to fit in the SRAM of the pico which is $264KB$, thus allowing for faster inference times.

3.4 Evaluation Metrics

The performance of the models is evaluated based on two main criteria: accuracy and latency.

Accuracy

The accuracy of the model is evaluated using the Mean Squared Error (MSE) metric (euclidean distance between the predicted location and ground truth).

Latency

Using the pico-tflmicro and pico-sdk libraries, the quantized models are deployed on the Pico, where the inference latency is measured. The latency is measured by running the model on a set of input tensors and averaging the time taken for multiple runs to get a reliable estimate. The latency is measured in milliseconds (ms).

4 Experimental Setup

This section describes the experimental setup used to evaluate the proposed methodology. All models were trained and evaluated for accuracy on an NVIDIA T4 GPU. Inference latency, however, was measured separately on a Raspberry Pi Pico.

4.1 Baseline Model

We use the MLP model used by Zhu et al. [10] as our baseline model. The model has five hidden layers with 256, 512, 1024, 512 and 256 neurons respectively and output layer of size 2, ReLU activation function, Adam Optimizer with learning rate of 1×10^{-4} for gradient descent. The model is trained for 15 epochs with batch size of 64. The input of this model is a 1×36 vector corresponding to the 36 RSS values. The model is also quantized using PTQ to 8-bit which reduces the model size of $5.1KB$ to $1.3KB$, thus can only be deployed on the Pico's flash memory.

Layer Type	Options
Fully Connected Layer	32, 64, 128, 256, 512, 1024, 2048
Activation Function	ReLU, LeakyReLU, ELU, Hardswish, Tanh, Sigmoid

Table 1: Search space configuration for MLP architectures, showing the available options for layer widths and activation functions. Default PyTorch parameters are used for all activation functions.

Layer Type	Options
2D Convolution	Channels: 16,32,64,128,256 Kernel Size: 3, 5 Stride: 0, 1 Padding: 0, 1
2D Batch Normalization	Channels: 16,32,64,128,256
2D Pooling	Type: Average, Maximum
Full Connected Layer	32, 64, 128, 256, 512
Activation Function	ReLU, LeakyReLU, ELU, Hardswish, TanH, Sigmoid

Table 2: Search space configuration for the simple CNN architectures, showing the available options for different layer types and activation functions. Default PyTorch parameters are used for all activation functions.

4.2 Neural Architecture Search

A search is performed for both MLP and CNN architectures on the clean datasets. Architectures found are then re-trained on both raw and clean datasets to compare its performance on both datasets.

MLP Search

The MLP search space encompasses architectures with a variable number of hidden layers ranging from 2 to 6, variable layer dimensions and activation functions. The possible activation functions are selected based on their compatibility with TFLite-micro. Table 2 provides a detailed overview of the search space.

ENAS strategy is used for the MLP search, where the supernet is trained for 50 epochs using the Adam optimizer with a learning rate of 1×10^{-4} and batch size of 64. The RL controller uses the Proximal policy optimization (PPO) algorithm with a Long Short-Term Memory (LSTM) model with hidden size of 64, learning rate of 1×10^{-4} and discount factor of 1.0. These parameters are the default values used by the NNI framework for the ENAS strategy.

. Top 10 architectures are selected based on the validation error and are then re-trained for 50 epochs with the same hyperparameters. Early stopping with a patience of 5 epochs is used to prevent overfitting, and the model from the best-performing epoch (based on validation loss) is saved and used for evaluation. The NAS is conducted on the clean dataset with a maximum quantized model size of $250KB$ for SRAM deployment and $1900KB$ for flash memory deployment, to compare the trade-offs between model size and performance. The input of this model is a 1×36 vector corresponding to the 36 RSS values, same as the baseline model.

CNN Search

We perform the search on two different search spaces:

• Cell based search space: This search space consists of normal and reduction cells, where each cell is a small sub-network that can be stacked to form a larger network, which are then followed by a few fully connected layers. The cell architecture is very similar to the one used in DARTS [24], but with a few modifications to make it compatible with TFLite-micro. The possible operations within a cell are: `max_pool_3x3`, `avg_pool_3x3`, `skip_connect`, `sep_conv_3x3`, `sep_conv_5x5`, `dil_conv_3x3` and `dil_conv_5x5`. Number of cells can be between 6 and 15, which are then followed by FC layers to flatten and result in the output shape of 1×2.

• Simple search space: This search space consists of basic layers like convolutional layers, pooling layers and FC layers.

DARTS strategy is used for the cell based search space, where the supernet and architecture parameters are trained for 30 epochs using the Adam optimizer with a learning rate of 1×10^{-3}, gradient clipping of 0.5 and batch size of 128. The architecture is trained using the AdamW optimizer with a cosine learning rate schedule, starting at 1×10^{-3} and decaying to 1×10^{-5} over the course of training. The weight decay is set to 3×10^{-4}.

The search space is penalized for the number of parameters to ensure the hardware constraints are met. The best architecture is selected based on the validation error and is then re-trained for 100 epochs with the same hyperparameters. For the simple search space, we use also DARTS strategy, with the same hyperparameters as above, expect the number of epochs for training the supernet and architecture parameters is 15. The best architecture is selected based on the validation error and is then re-trained for 50 epochs with the same hyperparameters.

The NAS is conducted on the clean dataset, same as the MLP search. For the search of cell based architectures, the maximum quantized model size is set to $200KB$, since the cell based architecture tend to use more memory during inference. Based on trail and error, we found that on average the cell based architecture use around $60KB$ of memory, so we set the maximum model size to $200KB$ to ensure that the architecture can fit in the memory of the Raspberry Pi Pico. In contrast, for the simple search space, the maximum quantized model size is set to $230KB$, since the simple architectures tend to use less memory during inference. For flash memory deployment the maximum quantized model size is set to $1800KB$ for both search spaces. The input shape for the CNN architectures is 6×6, which corresponds to the 36 RSS values reshaped into a grid.

4.3 Augmented Data

To evaluate how the models perform on different fingerprint data densities, we run the best models found in the previous searches on augmented data with densities of $8cm$, $4cm$ and $2cm$. We do not perform a new NAS for each density; instead,

Model size (KB)		5168
Model Size post quantization (KB)		1303
Positioning error (mm)	Raw	20.7
	Clean	14.2
Positioning error post quantization (mm)	Raw	21.7
	Clean	16.2
Inference latency (ms)		283

Table 3: Performance of the baseline MLP model

we reuse the top architectures discovered from the NAS on the clean dataset and retrain them. This approach focuses on testing the models' ability to handle varying data densities and on evaluating the effectiveness of the data augmentation methods introduced by Zhu et al. [10].

5 Evaluation

5.1 Baseline

We run the baseline model, which is a Multilayer Perceptron (MLP) with five hidden layers, on the raw and clean datasets. Since, it is MLP architecture, the input shape is $1x36$. We also quantize the baseline model to 8-bit and run it on the Pico. Table 3 shows the performance of the baseline model on both datasets and the inference latency on the Pico. The results indicate that the baseline model performs well on both datasets, but the clean dataset yields better performance in terms of error and model size. Post quantization, the model size is reduced significantly from 5168 KB to 1303 KB, making it feasible to run on the Raspberry Pi Pico. The error increases slightly after quantization both on raw and clean datasets

5.2 Search

MLP Search

The MLP search took around 2 hours to find top 10 architectures. Each architecture took around 2 minutes for re-training for each dataset.

5.3 CNN Search

The CNN cell based search took around 4 hours to find architectures for the normal and the reduction cell. Figure 6 show the architectures of the normal and reduction cells found by DARTS strategy with model size constraint for SRAM deployment. For the simple search space, the search took around 30 minutes to find the top architecture, which then took around 5 minutes for re-training.

5.4 Results

Table 5 summarizes the top-performing architectures identified through our experiments. The exact architectural details for each model are provided in Appendix A. Table 4 presents the performance of these models across raw, clean, and augmented datasets. We report both quantized and non-quantized accuracies, as well as inference latency measured on the Raspberry Pi Pico. All selected models outperform the baseline in terms of both positioning error and inference speed.

`MLP_SRAM` is the smallest model, with a size of $553KB$. It achieves a positioning error of $7.4mm$ on the clean dataset and boasts an exceptionally fast inference latency of $18ms$ on the Pico. Because it fits entirely within SRAM, it benefits from significantly faster inference compared to models that must run from flash memory.

`MLP_FLASH`, the largest MLP model at $1344KB$, achieves improved positioning accuracy due to its larger hidden layers, which can capture more complex patterns. However, this is not a general trend, more neurons do not always equate to better performance.

The best accuracy is achieved by the CNN models. `CNN_Simple_Flash`, for instance, attains a positioning error of $4.1mm$ on the clean dataset an impressive improvement over the baseline. Other simple CNNs also achieve competitive results. However, their inference latency is higher than MLPs due to the computational complexity of convolutional operations.

The cell-based CNN model, `CNN_Cell_Flash`, achieves a positioning error of $6.9mm$ slightly worse than the simple CNN but still better than both the MLPs and the baseline. Its increased complexity and deeper architecture result in higher memory usage and latency during inference.

After post-training quantization, all models show a substantial reduction in model size. `MLP_SRAM` compresses down to just $144KB$. In contrast, `CNN_Cell_SRAM` sees a less dramatic size reduction only about $1.3\times$ compared to an average $4\times$ reduction seen in other models, likely due to the structural complexity of the cell-based architecture.

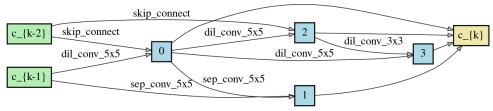
Quantization generally leads to a slight degradation in positioning accuracy, which is expected due to the reduced numerical precision of weights and activations. Still, the models perform well post-quantization. For example, `CNN_Simple_Flash` maintains a strong performance with a positioning error of $12.1mm$. The drop in accuracy varies across models, suggesting differing levels of quantization robustness. For instance, while `MLP_SRAM` experiences a twofold increase in error, `CNN_Cell_SRAM` shows a threefold increase highlighting that simpler models like MLPs are less sensitive to quantization than more complex CNNs.

5.5 Augmented Data

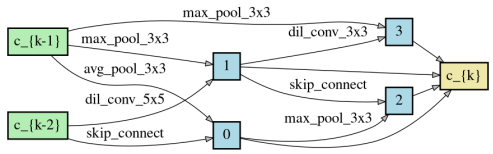
On average, the models perform better on the augmented data than on the raw dataset, indicating that the data augmentation methods are effective in improving the models' performance. Similarly, the positioning error increases slightly as the data density decreases, which is expected since the models are trained on denser data. In some cases, like the `MLP_Flash` model where the error on raw dataset is less than the error for augmented data and the `CNN_Cell_SRAM` model where the error on $8cm$ augmented data is less than the error on $4cm$ augmented data. These anomalies could be due to the randomness involved in the training or due to the models not being able to generalize well to the augmented data.

Name	Model Size (KB)	Model Size PQ (KB)	Positioning Error (mm)					Positioning Error PQ (mm)					Latency (ms)
			Raw	Clean	2cm	4cm	8cm	Raw	Clean	2cm	4cm	8cm	
Baseline	5168	1303	20.7	14.2	14.6	15.3	17.6	21.7	16.2	16.5	15.8	19.0	283
MLP_SRAM	553	114	10.0	7.4	10.4	14.6	15.8	20.1	16.9	15.6	17.0	18.9	18
MLP_Flash	1344	351	9.8	6.6	10.8	13.1	11.1	16.8	12.6	15.6	16.9	18.3	67
CNN_Simple_SRAM	548	155	8.2	4.9	6.3	6.9	6.7	14.1	12.3	12.8	13.8	16.2	84
CNN_Simple_Flash	5689	1446	6.4	4.1	5.3	5.7	6.0	19.6	12.1	12.2	12.4	13.6	334
CNN_Cell_SRAM	235	176	6.9	4.8	5.6	5.3	5.7	19.5	16.2	17.1	18.9	19.3	241

Table 4: Comparison of architectures found by NAS. PQ stands for Post Quantization. In Positioning error, the columns $2cm$, $4cm$ and $8cm$ represent the error on augmented data with densities of $2cm$, $4cm$ and $8cm$ respectively. Latency is inference latency measured on the Raspberry Pi Pico.



a) Normal cell



b) Reduction cell

Figure 6: Cell architectures Normal cell (a) and Reduction cell (b) found by DARTS strategy with model size constraint for SRAM deployment.

Model Name	**Search Space**	**Constraint**
MLP_SRAM	MLP	SRAM
MLP_Flash	MLP	Flash
CNN_Simple_SRAM	Simple CNN	SRAM
CNN_Simple_Flash	Simple CNN	Flash
CNN_Cell_SRAM	Cell CNN	SRAM

Table 5: Top architectures found from the different experiments.

6 Discussion and Limitations

Neural Architecture Search (NAS) proved effective in identifying models with superior positioning accuracy and lower inference latency on the Raspberry Pi Pico, highlighting its utility in optimizing neural networks under tight resource constraints. While traditional cell-based architectures commonly employed in state-of-the-art NAS frameworks were also explored, they did not outperform the simpler, directly searched architectures. This indicates that the positioning task considered here is relatively simple, and more complex NAS strategies may not be necessary. Nonetheless, NAS played a valuable role in adapting models to different hardware constraints, such as memory and inference latency, by discovering architectures that balanced accuracy, size, and latency. Post-training quantization yielded inconsistent results across models, some performing much worse than the others, suggesting room for improvement through advanced compression techniques like pruning, knowledge distillation, or quantization-aware training.

Data augmentation, though not outperforming the clean dataset (since it was derived from it), led to significantly better results than training on raw data and effectively reduced the manual burden of data collection. Moving forward, it would be worthwhile to explore larger and more expressive search spaces, though such efforts should be weighed against the computational cost, especially given the marginal gains observed for this relatively constrained task.

Furthermore, enhanced data preprocessing techniques could simplify the model design process, enabling even lightweight architectures to achieve high accuracy. As future datasets become more complex and representative of real-world deployment conditions, NAS is likely to become more impactful, uncovering diverse and robust architectures that outperform manual designs.

6.1 Limitations and Future Work

This research presents several limitations that warrant discussion. The NAS methodology employed focuses on discovering simple architectures for a relatively straightforward problem compared to state-of-the-art applications, making traditional NAS approaches not directly applicable to this domain. The search space is constrained to a limited set of layers and

hyperparameters, which may prove insufficient for identifying optimal architectures for the given problem. Furthermore, the search space is not exhaustive, potentially overlooking superior architectures that remain undiscovered by the NAS process, with the architectures found showing minimal performance differences. While hardware constraints provide valuable guidance for selecting architectures suitable for target devices, they simultaneously restrict both the search space and model performance capabilities.

Additionally, the dataset limitations present significant constraints, as it is restricted to the DenseVLC testbed, which may not adequately represent other VLP systems or diverse environments. The dataset generated is in a confined location, missing environmental factors such as interference, noise, and other real-world conditions that could impact the performance of the models in practical applications. This is not a limitation for the NAS process, but rather a limitation of the dataset, and it would be interesting to see how the models perform in a more realistic setting.

7 Responsible Research

We address the ethical considerations of our research, reproducibility of our results, and the use of generative AI tools in the writing process.

7.1 Ethical Considerations

The data collected for this research is from the DenseVLC testbed, which only contains measurements of RSS values and thus doesn't contain any sensitive or personal information. Also, VLP systems use LEDs with visible light, which doesn't pose any health risks to humans. However, the during data collection, rapid flashing of the LEDs may cause discomfort to some individuals.

7.2 Reproducibility

To ensure the reproducibility of our research, we have made all the code used in this research publicly available on Github[1]. Our code, uses open source libraries and frameworks, and is well documented for easy understanding and use.

7.3 Use of Generative AI

We have used generative AI tools, such as ChatGPT and Gemini, to assist in writing of this paper, in specific for tasks related to grammar and style checking. Prompts like *Please rewrite the following text to improve its clarity and style* were used to improve the quality of the text. All outputs from these tools were critically reviewed and evaluated. These tools were not used to generate any results.

8 Conclusions

In this paper, we used Neural Architecture Search (NAS) to find efficient MLP and CNN architectures for Visible Light Positioning (VLP) using Received Signal Strength data. Our models target the Raspberry Pi Pico and improve positioning accuracy by 50% compared to prior work by Zhu et al. [10], while achieving a low inference latency under $100ms$ on the Pico.

We also showed that our models maintain good performance when trained with augmented data, helping reduce the manual effort needed for data collection.

Our results show that NAS can effectively discover architectures that are well-suited for both the target hardware and the task. However, given the simplicity of the problem, hand-crafted architectures could likely achieve similar results. While NAS is not strictly necessary in this case, it remains valuable for more complex scenarios.

In future work, with more challenging datasets and environments, NAS could offer greater benefits by discovering diverse and higher-performing architectures that outperform manually designed models.

[1] https://github.com/Idkwhoami42/vlp-nas

References

[1] K. Chintalapudi, A. Padmanabha Iyer, and V. N. Padmanabhan, "Indoor localization without the pain," in *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 173–184. [Online]. Available: https://doi.org/10.1145/1859995.1860016

[2] D. Chen, K. G. Shin, Y. Jiang, and K.-H. Kim, "Locating and tracking ble beacons with smartphones," in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 263–275. [Online]. Available: https://doi.org/10.1145/3143361.3143385

[3] W.-H. Kuo, Y.-S. Chen, G.-T. Jen, and T.-W. Lu, "An intelligent positioning approach: Rssi-based indoor and outdoor localization scheme in zigbee networks," 07 2010, pp. 2754–2759.

[4] L. Barbieri, M. Brambilla, A. Trabattoni, S. Mervic, and M. Nicoli, "UWB Localization in a Smart Factory: Augmentation Methods and Experimental Assessment," *IEEE Transactions on Instrumentation Measurement*, vol. 70, p. 3074403, Jan. 2021.

[5] J. Torres-Sospedra, R. Montoliu, S. Trilles Oliver, O. Belmonte Fernández, and J. Huerta, "Comprehensive analysis of distance and similarity measures for wi-fi fingerprinting indoor positioning systems," *Expert Systems with Applications*, vol. 42, pp. 9263–9278, 12 2015.

[6] G. Seco-Granados, J. López-Salcedo, D. Jiménez-Baños, and G. López-Risueño, "Challenges in indoor global navigation satellite systems: Unveiling its core features in signal processing," *IEEE Signal Processing Magazine*, vol. 29, no. 2, pp. 108–131, 2012.

[7] A. Gradim, P. Fonseca, L. Alves, and R. Mohamed, "On the usage of machine learning techniques to improve position accuracy in visible light positioning systems," 07 2018, pp. 1–6.

[8] P. Du, S. Zhang, C. Chen, A. Alphones, and W.-D. Zhong, "Demonstration of a low-complexity indoor visible light positioning system using an enhanced tdoa scheme," *IEEE Photonics Journal*, vol. 10, p. 7905110, 08 2018.

[9] S. Zhang, P. Du, C. Chen, and W. Zhong, "3d indoor visible light positioning system using rss ratio with neural network," 07 2018.

[10] R. Zhu, M. Van den Abeele, J. Beysens, J. Yang, and Q. Wang, "Centimeter-level indoor visible light positioning," *IEEE Communications Magazine*, vol. 62, no. 3, pp. 48–53, 2024.

[11] J. Luo, L. Fan, and H. Li, "Indoor positioning systems based on visible light communication: State of the art," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2871–2893, 2017.

[12] L.-S. Hsu, D.-C. Tsai, H. M. Chen, Y.-H. Chang, Y. Liu, C.-W. Chow, S.-H. Son, and C.-H. Yeh, "Using received-signal-strength (rss) pre-processing and convolutional neural network (cnn) to enhance position accuracy in visible light positioning (vlp)," in *2022 Optical Fiber Communications Conference and Exhibition (OFC)*, 2022, pp. 1–3.

[13] J. Beysens, A. Galisteo, Q. Wang, D. Juara, D. Giustiniano, and S. Pollin, "Densevlc: a cell-free massive mimo system with distributed leds," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 320–332. [Online]. Available: https://doi.org/10.1145/3281411.3281423

[14] P. Ren, Y. Xiao, X. Chang, P.-y. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," *ACM Comput. Surv.*, vol. 54, no. 4, May 2021. [Online]. Available: https://doi.org/10.1145/3447582

[15] E. Liberis, Łukasz Dudziak, and N. D. Lane, "μnas: Constrained neural architecture search for microcontrollers," 2020. [Online]. Available: https://arxiv.org/abs/2010.14246

[16] C. Banbury, C. Zhou, I. Fedorov, R. M. Navarro, U. Thakker, D. Gope, V. J. Reddi, M. Mattina, and P. N. Whatmough, "Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers," 2021. [Online]. Available: https://arxiv.org/abs/2010.11267

[17] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," 2021. [Online]. Available: https://arxiv.org/abs/2103.13630

[18] L. Capogrosso, F. Cunico, D. S. Cheng, F. Fummi, and M. Cristani, "A machine learning-oriented survey on tiny machine learning," 2023. [Online]. Available: https://arxiv.org/abs/2309.11932

[19] Microsoft, "Neural Network Intelligence," 1 2023. [Online]. Available: https://github.com/microsoft/nni

[20] H. Benmeziane, K. E. Maghraoui, H. Ouarnoughi, S. Niar, M. Wistuba, and N. Wang, "A comprehensive survey on hardware-aware neural architecture search," 2021. [Online]. Available: https://arxiv.org/abs/2101.09336

[21] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," 2018. [Online]. Available: https://arxiv.org/abs/1707.07012

[22] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2017. [Online]. Available: https://arxiv.org/abs/1611.01578

[23] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," 2018. [Online]. Available: https://arxiv.org/abs/1802.03268

[24] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," 2019. [Online]. Available: https://arxiv.org/abs/1806.09055

[25] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," 2019. [Online]. Available: https://arxiv.org/abs/1808.05377

A Model Architectures

- **MLP_SRAM:**
  FC(36) → FC(256) + Hardswish → FC(256) + Hardswish → FC(256) + Tanh → FC(2) + Sigmoid
- **MLP_Flash:**
  FC(36) → FC(2048) + LeakyReLU → FC(64) + LeakyReLU → FC(2048) + LeakyReLU → FC(2) + Sigmoid
- **CNN_Simple_SRAM:**
  Conv2d(1, 32, kernel=1, padding=1) → BatchNorm2d(32) → ReLU →
  DepthwiseSeparableConv2d(32, 64, kernel=3) → BatchNorm2d(64) → Sigmoid →
  DepthwiseSeparableConv2d(64, 128, kernel=3) → BatchNorm2d(128) → Tanh → AvgPool2d(2) →
  Flatten → Linear(512, 256) → Tanh → Linear(256, 2) → Sigmoid
- **CNN_Complex_Flash:**
  Conv2d(1, 32, kernel=1, padding=1) → BatchNorm2d(32) → LeakyReLU →
  DepthwiseSeparableConv2d(32, 64, kernel=3) → BatchNorm2d(64) → LeakyReLU →
  DepthwiseSeparableConv2d(64, 128, kernel=3) → BatchNorm2d(128) → Tanh → MaxPool2d(2) →
  Flatten → Linear(128×2×2, 1024) → Tanh → Linear(1024, 512) → ReLU →
  Linear(512, 512) → ReLU → Linear(512, 256) → Hardswish → Linear(256, 2) → Sigmoid