TinyML-Based Adaptive Speed Control for Car Robot

A Comparative Approach

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

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

Abstract

This work investigates the feasibility of performing monocular depth estimation on highly resource-constrained hardware, specifically the Raspberry Pi Pico Zero microcontroller. In contrast to existing approaches that rely on large convolutional networks and high-performance devices, this study explores a set of custom lightweight encoder-decoder architectures, including one inspired by L-ENet, L-EfficientUNet, μPyD-Net, and an LSTM-μPyD-Net combination, designed to operate within strict memory limits. These models were trained on a preprocessed KITTI dataset, with either LiDAR depth maps or SGM (Semi-Global Matching) dense depth maps, and evaluated in terms of accuracy, model size, and real-time inference performance. Results demonstrate that meaningful depth prediction is achievable on microcontrollers, paving the way for low-cost autonomous navigation systems and broader applications of TinyML in embedded robotics, with SGM proving to be the best preprocessing technique, and the LSTM-μPyD-Net having the best accuracy when trained on the full Train split of the KITTI dataset.

1 Introduction

Monocular depth estimation, the process of inferring distance information from a single image, is critical in automotive perception systems, especially for autonomous navigation. Traditionally, this task has been tackled using large, resource-intensive neural networks or stereo vision setups [1, 2, 3]. However, such approaches are infeasible for microcontroller-class hardware like the Raspberry Pi Pico Zero, which has only 264 KB of RAM and lacks floating-point acceleration hardware.

This project proposes and evaluates several lightweight, quantized deep learning models tailored for efficient depth estimation on extremely resource-constrained embedded systems. The primary objective is to find the best trade-off between accuracy and efficiency on the Raspberry Pi Pico.

Multiple models were adapted from existing depth estimation networks, then compressed and quantized using TinyML techniques. The result is a comparative analysis of four optimized models and two preprocessing techniques, which are evaluated for their classification accuracy on binned depth maps (error of at most 25%, 56.25%, and 95.3125%), memory footprint, inference latency, and feasibility of deployment on Cortex-M0 microcontrollers.

A full deployment pipeline was also developed to convert trained TensorFlow models into C arrays compatible with edge inference engines such as TensorFlow Lite for Microcontrollers. The research offers an end-to-end framework for bringing depth-aware computer vision to severely constrained embedded systems.

The structure of the paper is as follows. First, related research and previous methods will be presented. Section 3 describes the methodology, featuring implementation details, architectures, and training. The experimental setup is then described in section 4, and results are highlighted. Section 5 represents the responsible research part, where the ethics of the paper are discussed. Then, discussion of the results is in order in section 6. Finally, section 7 presents the conclusions taken from this work, and what future works could build upon.

2 Related Work

Depth estimation from monocular input has received sustained attention across the fields of computer vision and robotics. In 2019, Godard et al. demonstrated the effectiveness of self-supervised learning approaches for monocular depth estimation, achieving competitive results while reducing the need for ground truth depth data [1]. Later, Peluso et al. introduced a method tailored for deployment on microcontroller units (MCUs), illustrating the potential for monocular depth inference in constrained environments [4]. Some of this research will build upon Peluso et al.'s, treating their results as a baseline for MCU deployment of depth perception models [4].

More recent models, including Depth Anything v2, DepthCrafter, MiDaS, Marigold, and Metric3D, have pushed the limits of depth estimation accuracy by leveraging larger datasets, transformer-based backbones, and powerful GPUs during inference [2, 3, 5, 6, 7]. While these approaches offer high precision, they are generally unsuitable for microcontroller devices due to their extensive memory and computation resource demands.

To address this gap, lightweight networks such as L-EfficientUNet [8] and L-ENet [9] were proposed. These models strike a balance between inference efficiency and accuracy, making them suitable for embedded deployment. Inspired by these designs, this work builds and evaluates reduced versions of such models, along with original architectures optimized from the ground up for the constraints of the Raspberry Pi Pico. Despite the improvements of pruning and quantization techniques, no recent study has performed a direct comparison of post compression monocular depth models on Cortex-M0 hardware. This research addresses that gap, providing a practical solution for model performance and deployability in ultra-low-power embedded systems. Moreover, this research shows the feasibility of using these depth perception systems as guidance for speed control, pairing the TinyML model with a PID-based speed controller.

3 Methodology

3.1 Overview

The adaptive speed control task requires two components. First, a way to control a vehicle's velocity, which is the vehicle's drivetrain, paired with a fine-tuned control loop. Second, a way to perceive surroundings, or rather, the distances to its surroundings. For this reason, the monocular depth perception task is considered most useful. Since communication between the speed controller and the depth perception task can be adapted without the use of machine learning with only the use of inter-frame relative velocity to obstacles, the depth estimation task is the focus of this research.

This project explores the design, training, and deployment of lightweight convolutional neural networks for monocular depth estimation on the Raspberry Pi Pico, a Cortex-M0 microcontroller with only 264KB of SRAM and no operating system. Due to this extremely limited memory and processing environment, several constraints had to be imposed on both the model architecture and input resolution. The final models are evaluated not only for accuracy but also for practical deployability, including conversion into C arrays and real-time inference on the Pico itself.

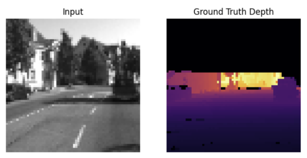
Initial experiments explored architectures based on MobileNetV1 and other lightweight backbones; however, the memory footprint and intermediate tensor sizes made these infeasible for deployment. As a result, custom architectures were developed from scratch, inspired by efficient segmentation and depth networks such as L-EfficientUNet and L-ENet, or pyramid architectures such as μPyD-Net, and designed with embedded deployment in mind [4, 8, 9, 10]. Four models were successfully adapted to run within the memory limits of the Pico. A fifth model based on binarized neural networks (BNNs) was implemented and trained, achieving a test accuracy of 58% in a classification style depth prediction task [11]. However, due to time constraints and incompatibility between TensorFlow Lite and the binary activation functions used, the BNN model was excluded from final deployment and moved to the future recommendations section.

3.2 Dataset and Preprocessing

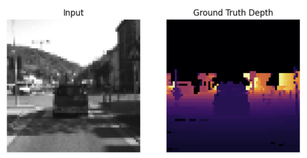
The KITTI dataset was selected as the basis for training due to its high-quality paired RGB and LiDAR depth data [12]. Each raw image from the dataset is first center-cropped to a square aspect ratio and then resized to a resolution of 64×64 pixels. This intermediate resolution was chosen to balance between retaining spatial detail and allowing for efficient further downsampling during deployment. The final model input resolution is 64×64, and images are converted to grayscale to reduce the input dimensionality and comply with SRAM constraints on the Raspberry Pi Pico.

For depth maps, two distinct preprocessing strategies were used. Following the first strategy, depth maps were generated by projecting LiDAR point clouds into the image plane using the provided calibration matrices. Points behind the camera or outside the field of view were filtered out, and sparse depth maps were formed by assigning projected depth values to valid image coordinates. To mitigate sparsity and create dense training targets, morphological dilation and nearest-neighbor interpolation were applied only in regions of sufficient point density. The resulting depth maps were clipped to a maximum range of 80 meters, log-normalized for training stability, and saved as NumPy arrays. The left camera input and the associated dense depth map computed using calibration data can be seen in Figure 1.

The second strategy generated dense depth maps without needing to use interpolation by leveraging the KITTI dataset's use of two distinct cameras simultaneously. As depth is measured in respect to the vehicle's left camera within the



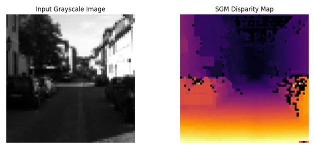
(a) Pair 1



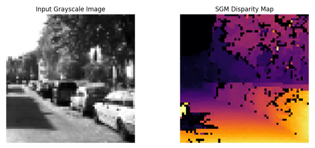
(b) Pair 2

Figure 1: Dense depth maps computed using interpolation of LiDAR data.

KITTI dataset, by applying SGM (Semi-Global Matching) to the frames obtained by the left and right cameras, a dense depth map can be obtained without interpolation or knowledge loss. Defunct pixels, where depth could not be computed, are marked with an invalid value, in order to be later ignored by the loss function. The left camera input and the associated depth map using SGM can be seen in Figure 2.



(a) Pair 1



(b) Pair 2

Figure 2: SGM preprocessing using stereo pairs, where pixels in dark blue are outliers [13].

On-device preprocessing was also required to adapt the ArduCam SPI module's native 320×320 resolution to the model's expected 64×64 input. This resizing is performed in C using nearest-neighbor interpolation for its computational simplicity and negligible memory overhead. This design enables real-time, low-latency adaptation of camera input without additional hardware or significant processing burden on the microcontroller.

3.3 Model Architectures

Two of the final models used in this project are compact encoder-decoder style networks designed for dense prediction at a resolution of 64×64 pixels. The model input was limited to grayscale images in this resolution due to SRAM constraints. The other two models rely on a pyramid architecture, and one also implements an LSTM, relying on sequential data in batches of 3 frames each instead of one image at a time. At the end of each model, Sigmoid activation is used in the case of the preprocessed LiDAR maps, and no activation is used in the case of SGM preprocessing.

L-EfficientUNet

The first architecture used in this study is based on L-EfficientUNet [8], a highly compact encoder-decoder network that achieves a strong trade-off between parameter efficiency and depth estimation accuracy.

L-EfficientUNet follows a traditional U-Net structure, and its architecture can be observed in Figure 3. It replaces standard convolutional layers with depthwise separable convolutions to drastically reduce the number of parameters and operations. The encoder consists of a sequence of blocks, each composed of a depthwise separable convolution followed by a max-pooling operation. These blocks progressively reduce spatial resolution while increasing feature depth. The decoder mirrors this structure by using bilinear upsampling and concatenating feature maps with their corresponding encoder outputs via skip connections, followed again by depthwise separable convolutions to refine spatial detail [8].

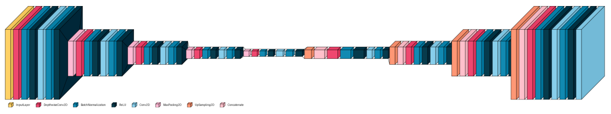


Figure 3: L-EfficientUNet Architecture [8]

The bottleneck layer, situated at the lowest resolution (4×4), processes high-level features with expanded depth before decoding. The final output is produced via a 1×1 convolution with sigmoid activation to generate a single channel depth map normalized between 0 and 1. Like the other models, L-EfficientUNet operates on grayscale 64×64 images as input.

Its lightweight design and use of efficient convolutions make it exceptionally well-suited for inference on microcontrollers with tight memory limits such as the Raspberry Pi Pico.

L-ENet Inspired Architecture

The second model explored in this work is a custom lightweight architecture inspired by L-ENet, originally proposed as a real-time semantic segmentation network for resource-constrained devices. While the original L-ENet was designed for semantic segmentation at higher resolutions, this adapted version preserves its structural philosophy while scaling it down to operate within the SRAM and flash memory constraints of the Raspberry Pi Pico.

This encoder-decoder model features four downsampling stages, leading to a compact 4×4 latent feature representation, and four corresponding upsampling steps to restore the resolution to 64×64. Skip connections after each major downsampling stage help retain spatial fidelity during reconstruction.

The architecture begins with an initial block that combines a standard convolution and a max-pooling operation to quickly reduce the spatial resolution while increasing feature depth. This is followed by a series of bottleneck modules, which consist of depthwise separable convolutions and residual connections. These modules can either downsample or upsample spatial resolution, depending on the position within the network.

Unlike the original L-ENet, which includes asymmetric and dilated convolutions, this version simplifies each bottleneck to a single depthwise separable convolution with stride control, followed by batch normalization and ReLU6 activation. Furthermore, instead of transposed convolutions used in L-ENet's decoder, bilinear upsampling is combined with residual skip connections, further reducing the memory footprint. The architecture of this model is showed in Figure 4.

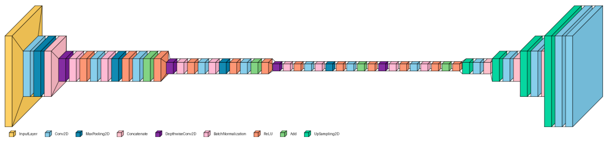


Figure 4: L-ENet Architecture [9]

Despite being based on a more complex architecture, this model fits within the constraints of the Raspberry Pi Pico after structural pruning, offering a balance between computational efficiency and prediction accuracy.

μPyD-Net

The third model evaluated in this study is μPyD-Net, a lightweight, pyramid-based convolutional neural network introduced by Peluso et al. in 2022 [4]. Designed specifically for ultra-low-power microcontroller platforms, μPyD-Net enables monocular depth estimation using extremely low-resolution grayscale inputs (as small as 32×32 or 48×48), making it ideal for edge AI applications with strict memory and power constraints. In this research, the model was also tested at a resolution of 64×64 pixels.

The architecture follows a shallow encoder-decoder design structured around a three-level pyramidal representation, as can be seen in Figure 5. The encoder consists of six 3×3 convolutional layers interleaved with leaky ReLU activations ($\alpha = 0.125$), gradually reducing the spatial resolution while increasing feature depth. Each encoder level feeds into a corresponding decoder block composed of three convolutional layers and an upsampling operation via 2×2 transposed convolutions. The final output is a coarse inverse depth map produced at the input resolution.

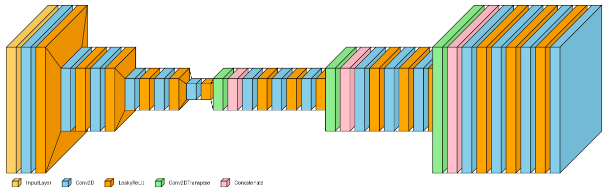


Figure 5: μPyd-Net Architecture [4]

This model was trained by Peluso et al. using SGM pseudo-ground truths instead of LiDAR depth maps [4], and the same methodology was also used by this research, adapting other models to the preprocessing technique also used by Tosi et al., following the algorithm of H. Hirschmuller [14, 13]. Due to μPyD-Net's structure, only this data was suitable for training, as the use of LiDAR depth maps showed minimal results.

Temporal-μPyD-Net

Building on the efficient design of μPyD-Net, the fourth model explored in this study is a temporal extension that integrates a convolutional recurrent layer to enable video-based depth estimation. This architecture, referred to as Temporal-μPyD-Net, combines the compact pyramidal encoder-decoder design of μPyD-Net with a simulated ConvLSTM2D unit through a Lambda layer, allowing the network to exploit temporal continuity across short image sequences. By modeling motion-aware features, it aims to improve depth prediction stability and accuracy in dynamic scenes.

Like the base model, the encoder processes grayscale inputs at low resolutions (64×64) using a sequence of lightweight convolutional blocks. However, in this case, each input is a short temporal window of consecutive frames. The encoder processes each frame independently through time-distributed convolutional layers, preserving spatial abstraction while maintaining the temporal structure of the input sequence.

At the bottleneck, the simulated ConvLSTM2D layer aggregates the temporally encoded features into a single representation, through a mean operation. This recurrent unit captures temporal correlations and motion cues across the sequence, outputting a refined latent feature map suitable for decoding. The decoder structure remains largely unchanged from μPyD-Net, consisting of three upsampling stages with skip connections. However, skip connections are drawn specifically from the last frame in the sequence, ensuring consistency with the temporal aggregation step. The reason why an actual LSTM layer was not used is because of TFLite Micro's lack of LSTM support as of today.

The final output is a single-channel inverse depth map, corresponding to the last frame of the input sequence. By integrating temporal reasoning into the μPyD-Net framework without significantly increasing its computational footprint, this architecture serves as a promising direction for lightweight, temporally-aware depth estimation on embedded systems. The architecture of this hybrid model is shown in Figure 6.

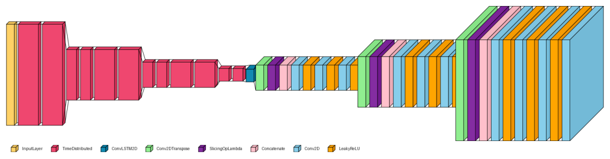


Figure 6: Original LSTM-μPyd-Net Architecture

A comparison of the number of parameters and whether the models fit within the Pico's SRAM post-quantization is presented in the Table 1.

Model	Params	SRAM Fit
L-EfficientUNet	124,328	Yes
L-ENet (full)	504,845	No
L-ENet (adapted)	25,493	Yes
μPyD-Net	130,073	Yes
LSTM-μPyD-Net	203,939	Yes
BNN	240,208	Yes*

Table 1: Summary of model characteristics (* - SRAM fit not verified due to TensorFlow Lite incompatibilities)

3.4 Training Strategy

Each model was trained using supervised learning with either mean squared error as the primary loss function, or reverse Huber loss [15]. The models were evaluated on a held-out test set using pixel-wise comparison of the predicted and ground truth depth values. Training proceeded for up to 100 epochs, with early stopping based on validation loss to prevent overfitting. Where applicable, quantization-aware training was applied to simulate the final deployment conditions. After training, the models were converted to TensorFlow Lite format, quantized to INT8, and further processed into C++ arrays suitable for embedding directly in microcontroller firmware. The models were then compiled using a Pico-specific version of TensorFlow Lite Micro [16].

While the binarized neural network achieved acceptable results during workstation testing, deployment was not possible due to the presence of unsupported operations in the TensorFlow Lite runtime. As a result, it is considered as part of future work.

3.5 Deployment and Inference

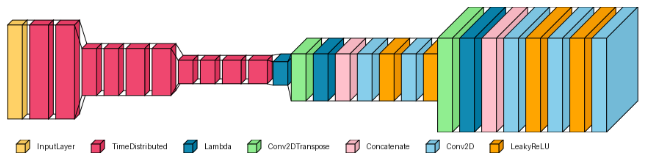
Final models were exported as C++ files containing 8 bytes aligned arrays of quantized weights, using TensorFlow Lite Micro as the runtime. The firmware implementation on the Raspberry Pi Pico handled real-time camera input, grayscale conversion, downsampling, inference, and output logic. The Pico's total SRAM usage was kept within the device's limit by reducing intermediate feature map sizes and aggressively limiting the number of filters used in convolutional layers.

To translate model output into control behavior, predicted depth maps were processed in firmware and passed into a simple decision-making routine. The objective was to enable the robot to interpret its environment and respond in real time, despite severe compute and memory constraints. This was done by examining the center of each image, and comparing an expected depth to an actual one. If it differed, it meant an object was encountered, and action had to be taken accordingly.

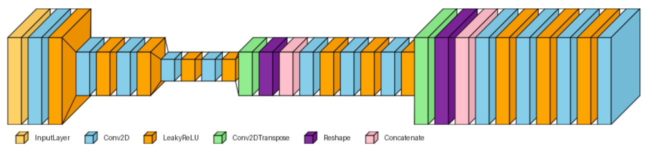
3.6 Control System

A PID (Proportional Integral Derivative) controller was implemented to convert the predicted depth outputs into motor control commands. The controller regulates forward velocity and turning direction based on the presence and location of nearby obstacles. The proportional term allows for immediate response to detected obstacles, the integral term helps correct cumulative drift in trajectory, and the derivative term dampens oscillations caused by noisy predictions.

This controller was tuned through empirical testing and implemented entirely on the Pico to preserve autonomy and eliminate external computation. By using depth maps as the control input, the system is able to adapt its speed in real time using only monocular vision and embedded computation.



Rescaled LSTM-based architecture



Rescaled μPyD-Net architecture

Figure 7: Rescaled Architectures for Raspberry Pi Pico

4 Experimental Setup and Results

4.1 Experimental Setup

The primary objective of the evaluation was to determine the trade-off between depth estimation accuracy and computational efficiency of lightweight neural networks when deployed on ultra-low-power microcontrollers, particularly the Raspberry Pi Pico. The experiments were conducted using a GPU-enabled development machine (NVIDIA RTX4060) for training and validation. Each model was then converted to TensorFlow Lite format, post-quantized to 8-bit fixed-point precision, and prepared for deployment on the Pico using the pico-tflmicro framework [16].

Input data consisted of paired grayscale and SGM depth maps derived from the KITTI dataset, preprocessed into grayscale 32×32 images and normalized dense depth targets, or non-normalized inverse SGM dense depth targets. A total of 22600 paired samples were used, while another 697 were used for testing, following the concept of an eigen split, as defined by Eigen et al. [17, 1]. Ground truth labels were clipped between 0 and 80 meters and normalized to a [0, 1] range in the case of depth maps, or had their unknown pixels masked in the case of SGM, remaining unnormalized. Data augmentation during training included random horizontal flips and targeted duplication for class imbalances, as most frames showed an open road with no other vehicles.

All models were trained using the Adam optimizer with an initial learning rate of 0.001 or 0.0001, a batch size of 8 or 16, and early stopping after 10 validation epochs with no improvement. Either Mean Squared Error (MSE) or reverse Huber (berHu) were used as the primary loss functions [15, 18], while Mean Absolute Error (MAE) and threshold-based accuracy ($\delta < 1.25$, $\delta < 1.25^2$, and $\delta < 1.25^3$) were tracked as secondary metrics. Threshold-based accuracy is a standard metric for regression tasks such as the one presented in this research, where δ represents the value obtained after calculating the maximum of the division between ground truth value and predicted value, or vice-versa. Therefore, the value of delta shows a scaled error, and by calculating how many pixels have a δ value under a certain threshold, we can define an accuracy metric.

4.2 Testing Procedure

Each model's predictions were evaluated on the reserved test set. Performance was measured in terms of Delta-accuracy. The output depths were rescaled to meters and compared to ground truth. Inference was unachievable on the Raspberry Pi Pico using a 32×32 resolution, even when using grayscale images. Therefore, images had to be downscaled to 32×32, saved to C++ arrays, and used for inference instead. Models also had to be retrained with this new resolution, using the same hyperparameters mentioned above, and the same architectures.

For on-device testing, the robot was placed on a flat table surface in a controlled indoor environment. Obstacle objects (books, boxes) were slowly moved into the robot's path while a real-time feed from the ArduCam was processed by the depth estimation model. A hard-coded threshold was used to classify dangerous proximity and trigger stop maneuvers, using the outputted SGM maps' depth bins as reference, as actual depth predictions would have to be scaled anyway, given KITTI was captured on a road environment, while the robot is no more than 15 centimeters (cm) tall. For reference, the robot used for testing is present in Figure 8.

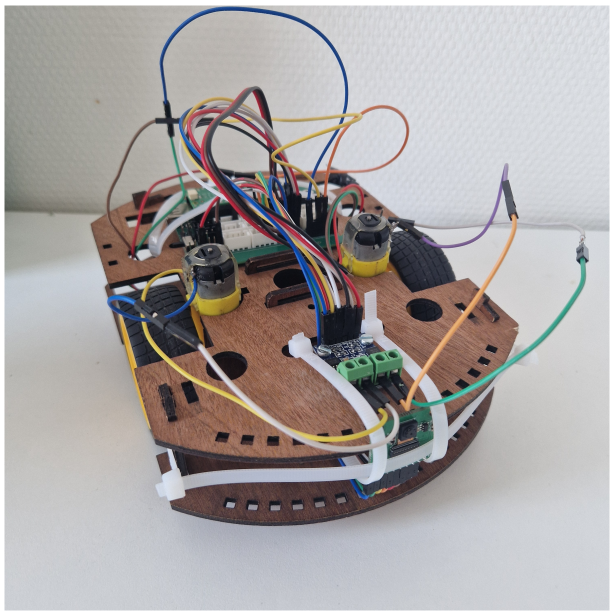


Figure 8: Robot with ArduCam in front and Raspberry Pi Pico

Given the fact that images had to be downscaled, the μPyD-Net models' architecture also changed slightly, to reflect the smaller data size. These can now be seen in Figure 7. The change is mostly related to the removal of the final decoder layer, as to not upsample once again to 64×64, and instead just leaving the resolution at 32×32. This also resulted in a small size decrease, as well as a small speedup. These are, however, barely visible after quantization, or when running inference on the Pico.

4.3 Results

The results for the four models, obtained during off-device testing on the workstation, are shown in Table 2. These are used as a metric for accuracy, given the fact computation stays the same. It is also important to mention that these are the results models get after full int8 quantization, on the entire KITTI test set [17].

Model	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$	Size (Pi Pico, KB)
L-EfficientUNet	54.40%	70.08%	80.77%	178KB
L-ENet	55.88%	73.24%	83.35%	49KB
μPyD-Net	74.32%	83.95%	88.44%	157KB
LSTM-μPyD-Net	74.38%	83.68%	88.40%	170KB

Table 2: Model Evaluation Results (GPU Workstation, 64×64 resolution)

One might notice that there are some differences in results from Peluso et al.'s research, namely a slight difference in obtained threshold accuracy [4]. Whereas they reported a $\delta < 1.25$ accuracy of 73.1% for 48×48 images, this paper reports an accuracy of 74.32% for the new image. As stated, the difference in the results between the paper of Peluso et al. and this work comes from the difference of input resolution (64×64 versus 48×48), and full integer quantization methods.

Results for the 32×32 architecture for his model are much closer in line with those reported by Peluso et al. [4]. Due to the much higher accuracy and lower memory constraints, only the μPyD-Net and original LSTM-μPyD-Net were used in the final tests. The accuracies on the Raspberry Pi Pico are reported in Table 3, as well as the final .tflite file sizes.

Model	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$	Size (Pi Pico, KB)
μPyD-Net	69.91%	80.99%	86.60%	75KB
LSTM-μPyD-Net	69.95%	81.05%	86.69%	74KB

Table 3: Model Evaluation Results (Raspberry Pi Pico, 32×32 resolution, full int8-quantization)

For the preprocessing method using dense interpolated LiDAR depth maps, the models seemed ineffective. They were unable to pick up on spacial cues, and only overfit on empty roads, even after solving the class imbalance, when they started predicting noise. This could be due to the models getting stuck in a local minimum due to the nature of MSE, bad weight initialization (randomization-dependent), or simply interpolated data not being clear enough. Therefore, the models failed to achieve over 60% accuracy for $\delta < 1.25$, or over 80% for $\delta < 1.25^2$, and although their accuracies may seem decent at first, they are overestimated because of the noisy prediction randomly being in the correct range, by only predicting the bottom of the image being close and the middle far, the rest being a gradient between the two.

However, the models were much more effective when used with the combination of SGM depth maps and reverse Huber loss (berHu), which provides smoother gradients and helps avoid local minima. Moreover, SGM depth maps are not interpolated, masking unavailable pixel depths as invalid instead of interpolating a depth.

For reference, we showcase some side-by-side predictions for all models below, in Figures 9.

Finally, for the two models used for Pico inference, two more metrics were especially important, namely inference time and SRAM usage out of the allocated tensor arena. Both models were given access to an arena of around 80kB of SRAM from the Pico to use, given the other processes active on the Pico, namely the PID controller and post-processing. For this to be possible, as mentioned in section 4.2, the inputs had to be lowered to a 32×32 resolution. Moreover, skip connections, that created implicit reshapes, had to be explicitly mentioned for the tflite micro pico runtime environment to correctly compute them [16]. Since the runtime requires static memory allocation, it will try to allocate enough SRAM to store all intermediate results, and never frees them. This is a strong limitation of the environment, and also the reason why inputs had to be downscaled. The results of running on the Pico can be seen in Table 4.

It should also be mentioned that the TFLite Micro library needed to be modified for the Temporal model to run, namely, the dimension limit for the Strided Slice operation needed to be changed from 4 to 5. The change had no side-effects since the rest of the code is generalized to n dimensions, and the choice for a maximum of 4 seems to be arbitrary.

Model	SRAM footprint (kB)	Inference time (seconds)
μPyD-Net	55.724	2.6
LSTM-μPyD-Net	90.044	3.8

Table 4: Model Evaluation Results (Raspberry Pi Pico, 32×32 resolution, full int8-quantization)

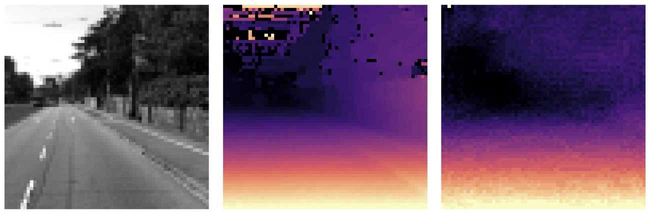
Given these results when evaluating on the Raspberry Pi Pico, it may not be an obvious choice whether a temporal model is still a feasible implementation choice. Both versions perform well on the Pico, and a minimal change in accuracy can be observed. The difference comes in SRAM footprint and inference time, where the normal μPyD-Net outperforms the new LSTM-based implementation. This is because of larger tensor sizes and more operations, as well as the triple size in input, since the LSTM takes three input images at a time. Seeing as this does not triple memory usage, when used at scale, the LSTM-based network might outperform the μPyD-Net at higher resolutions, when the size of the model itself weighs in more than input size [4]. It is also important to note the higher quality of results that the temporal model produces, offering more smooth and less noisy results.

5 Comparison of Results

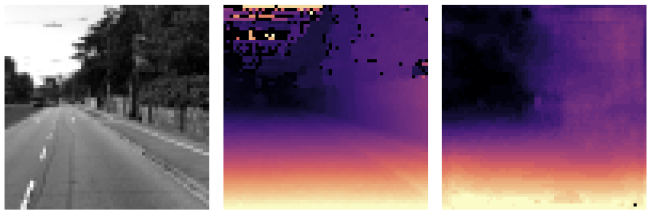
To evaluate the contribution of different design choices to the final performance, a comparison was conducted, analyzing the impact of the supervision signal, the loss function, and the learning rate.

Supervision Signal: LiDAR vs. SGM

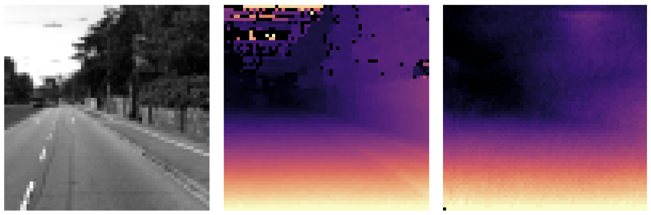
Sparse LiDAR ground truth was compared with dense depth maps generated via Semi-Global Matching (SGM)



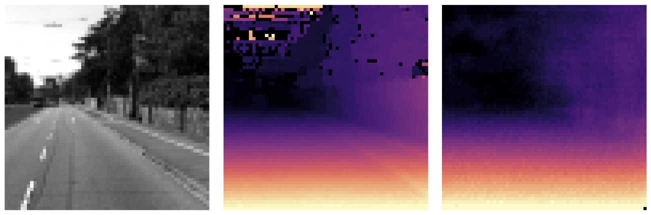
(a) L-EfficientUNet Predictions



(b) L-ENet Predictions



(c) μPyD-Net Predictions



(d) LSTM-μPyD-Net Predictions

Figure 9: Predictions from four models using SGM preprocessing [13].

as supervisory signals [13]. While LiDAR offers highly accurate depth values, its sparsity limits the model's ability to generalize to dense output. In contrast, SGM provides a denser signal, although noisier. Empirically, models trained with SGM supervision consistently outperformed those trained with LiDAR, confirming the benefit of dense supervision in this setting, confirming the findings of Peluso et al. [4].

Loss Function: MSE vs. berHu

The choice of loss function was then evaluated by comparing the standard Mean Squared Error (MSE) loss with the reverse Huber (berHu) loss. Training with berHu loss led to faster convergence and better final performance. This improvement can be attributed to berHu's robustness to outliers while maintaining sensitivity to small errors, an advantageous property when learning from noisy depth maps such as those produced by SGM [18].

Learning Rate: $1\mathrm{e}{-3}$ **vs.** $1\mathrm{e}{-4}$

Lastly, two learning rates were compared: $1\mathrm{e}{-3}$ and $1\mathrm{e}{-4}$. Models trained with a learning rate of $1\mathrm{e}{-4}$ demonstrated more stable training and better generalization, while $1\mathrm{e}{-3}$ frequently led to oscillations or suboptimal plateaus in performance.

In summary, these experiments show that using SGM as supervision, employing the berHu loss, and selecting a learning rate of $1\mathrm{e}{-4}$ are all critical to achieving optimal results. The ablation results highlight the importance of careful configuration in dense depth estimation pipelines.

6 Responsible Research

6.1 Societal Impact

This work contributes to the ongoing advancement of affordable autonomous systems by demonstrating that monocular depth estimation can be performed effectively on extremely resource-constrained devices. By showing that models under 60KB can still achieve reliable inference results, the research highlights the feasibility of deploying computer vision-based navigation systems on widely available microcontrollers such as the Raspberry Pi Pico. This could lower the entry barrier for low-cost robotics and facilitate broader adoption of embedded intelligence in fields such as smart mobility, warehouse automation, and education.

The findings are particularly relevant for the automotive industry, where many existing embedded systems lack the computational capacity to run modern perception models. Integrating TinyML models into existing generic nodes, without the need for additional hardware accelerators, has the potential to reduce system complexity and cost while still enabling safe, vision-based obstacle detection and navigation.

6.2 LLM Disclosure

Large Language Models (LLMs), specifically OpenAI's ChatGPT, were used throughout the research process for technical proofreading, refining scientific writing. All final decisions regarding model design, experimentation, and evaluation were made independently by me.

7 Discussion

7.1 Interpretation of Results

The objective of this research was to evaluate whether lightweight depth estimation models can operate effectively within the memory and computational limits of the Raspberry Pi Pico, thereby addressing the research question: *"What is the post-compression efficiency of TinyML depth perception models when run on the Raspberry Pi Pico?"*

The results confirm that meaningful depth predictions are achievable even with heavily compressed neural network models operating on 32×32 grayscale inputs. Two of the four selected architectures successfully compiled to TensorFlow Lite and met strict memory constraints, while maintaining reasonable test-set accuracy when evaluated on a GPU-enabled workstation.

Moreover, they prove that the combination of preprocessing and loss used by Peluso et al., namely computing SGM depth maps and using reverse Huber loss, are the ideal combination for maximizing delta accuracy for multiple models and architectures [4].

Finally, the temporal nature of the recordings enabled the implementation of an LSTM, that enhanced the smoothness of predictions over time, but also increasing inference time and using more of the Pico's SRAM.

7.2 Limitations

While the results are promising, this study encountered several limitations. First, due to hardware and environmental constraints, it was not feasible to collect a custom dataset tailored specifically for this application. Instead, the KITTI dataset was used for training and evaluation. Though robust, KITTI is largely composed of outdoor automotive scenes and may not generalize perfectly to indoor testing scenarios or environments with substantially different lighting or scale. In the future, the use of a simulated dataset such as the CARLA dataset [19], might prove beneficial.

Second, while a binarized neural network (BNN) was initially considered and partially implemented, time constraints and quantization framework limitations prevented its full integration and on-device deployment. Preliminary experiments on the workstation showed that the BNN achieved approximately 58% test accuracy after training with quantization aware methods using LiDAR interpolated depth maps as supervision outputs, confirming its feasibility but highlighting the need for further tuning and integration work.

Finally, given that the Raspberry Pi Pico Zero has limited multithreading capabilities and misses an FPU (Floating Point Unit), it fails to fully utilize the capabilities of the deployed models, given high inference times. Therefore, this study recommends against using such a device, and rather upgrading to a faster Cortex-M processor, such as the Cortex-M4, or the Cortex-M7, given that this research's applications are in autonomous driving. However, further fine-tuning and research may prove that the Raspberry Pi Pico Zero is also enough to safely perform such a task.

7.3 Broader Implications

Despite the outlined limitations, the findings contribute meaningfully to the field of embedded machine learning, particularly in the automotive robotics domain. The study demonstrates that full-stack monocular depth estimation can be compressed into a sub-60 KB binary and executed within kilobytes of SRAM, paving the way for affordable, vision-only navigation systems. This opens new pathways for integrating TinyML perception modules into existing generic microcontroller-based automotive platforms, such as driver-assistance systems, automated parking modules, or robotic delivery agents.

By providing a clear comparison of depth models under Pico constraints, this research serves as a foundation for further work in model search, hardware-aware compression, and adaptive depth estimation in edge environments.

8 Conclusions and Future Work

This study explored the viability of deploying lightweight monocular depth estimation models on the extremely resource-constrained Raspberry Pi Pico microcontroller. By leveraging techniques such as depthwise separable convolutions, bottleneck modules, careful architectural pruning, and simulated Long-Short Term Memory (LSTM), four distinct convolutional neural networks were evaluated. All models adhered to strict memory limitations, using grayscale inputs at 32×32 resolution, and achieved inference-ready sizes suitable for embedded deployment on the Raspberry Pi Pico.

In summary, this work contributes:

- A comparative analysis of four compact depth estimation models tailored for Cortex-M0+ hardware.
- Two complete preprocessing and training pipelines for KITTI-based monocular depth learning at 64×64 and 32×32 resolution.
- Evidence that depth perception for autonomous robotics can be achieved using sub-200 KB models on microcontrollers without specialized hardware accelerators.

When it comes to future work, it is strongly recommended that these models are also tested on a better board, that can better leverage their architecture. Even just upgrading to a device such as the Raspberry Pi Pico Two might make the work more than feasible. The use of pure CMSIS-NN instead of a library like TFLite Micro may also be beneficial, while also enabling the use of device-specific optimizations.

References

[1] Clement Godard, Oisin Mac Aodha, Michael Firman, and Gabriel J. Brostow. Digging into self-supervised monocular depth estimation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[2] Lihe Yang, Bingyi Kang, Zilong Huang, Zhen Zhao, Xiaogang Xu, Jiashi Feng, and Hengshuang Zhao. Depth anything v2. *Advances in Neural Information Processing Systems*, 37:21875–21911, 2024.

[3] Mu Hu, Wei Yin, Chi Zhang, Zhipeng Cai, Xiaoxiao Long, Hao Chen, Kaixuan Wang, Gang Yu, Chunhua Shen, and Shaojie Shen. Metric3d v2: A versatile monocular geometric foundation model for zero-shot metric depth and surface normal estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

[4] Valentino Peluso, Antonio Cipolletta, Andrea Calimera, Matteo Poggi, Fabio Tosi, Filippo Aleotti, and Stefano Mattoccia. Monocular depth perception on microcontrollers for edge applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(3):1524–1536, 2022.

[5] Wenbo Hu, Xiangjun Gao, Xiaoyu Li, Sijie Zhao, Xiaodong Cun, Yong Zhang, Long Quan, and Ying Shan. Depthcrafter: Generating consistent long depth sequences for open-world videos. *arXiv preprint arXiv:2409.02095*, 2024.

[6] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. *IEEE transactions on pattern analysis and machine intelligence*, 44(3):1623–1637, 2020.

[7] Bingxin Ke, Anton Obukhov, Shengyu Huang, Nando Metzger, Rodrigo Caye Daudt, and Konrad Schindler. Repurposing diffusion-based image generators for monocular depth estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9492–9502, 2024.

[8] Xiangyu Liu, Junyan Chen, Yan Zhou, Yuexia Zhou, Jinhai Wang, Xiaoquan Ou, and Haotian Lei. *L-EfficientUNet: Lightweight End-to-End Monocular Depth Estimation for Mobile Robots*, pages 394–408. 10 2023.

[9] Sangwon Kim, Jaeyeal Nam, and Byoung Chul Ko. Fast depth estimation in a single image using lightweight efficient neural network. *Sensors*, 19:4434, 10 2019.

[10] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[11] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[12] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[13] H. Hirschmuller. Accurate and efficient stereo processing by semi-global matching and mutual information. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 807–814 vol. 2, 2005.

[14] Fabio Tosi, Filippo Aleotti, Matteo Poggi, and Stefano Mattoccia. Learning monocular depth estimation infusing traditional stereo knowledge. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9799–9809, 2019.

[15] Laurent Zwald and Sophie Lambert-Lacroix. The berhu penalty and the grouped effect. *arXiv preprint arXiv:1207.6868*, 2012.

[16] Raspberry Pi Foundation. pico-tflmicro: Tensorflow lite for microcontrollers on raspberry pi pico. https://github.com/raspberrypi/pico-tflmicro, 2023. Accessed: 2025-06-01.

[17] David Eigen, Christian Puhrsch, and Rob Fergus. Depth map prediction from a single image using a multi-scale deep network. *Advances in neural information processing systems*, 27, 2014.

[18] Iro Laina, Christian Rupprecht, Vasileios Belagiannis, Federico Tombari, and Nassir Navab. Deeper depth prediction with fully convolutional residual networks. In *2016 Fourth international conference on 3D vision (3DV)*, pages 239–248. IEEE, 2016.

[19] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.