ESP32-WROOM Neural Network Compression

1st Dominik Kornak
Electrical and Computer Engineering
University of Illinois at Chicago
Chicago, IL, USA
kornakdominik@gmail.com

2nd Damien Karpen
Electrical and Computer Engineering
University of Illinois at Chicago
Chicago, IL, USA
dkarp@uic.edu

3rd Anand Pudi
Electrical and Computer Engineering
University of Illinois at Chicago
Chicago, IL, USA
apudi2@uic.edu

4th Alp Berke Ardic
Electrical and Computer Engineering
University of Illinois at Chicago
Chicago, IL, USA
aardic2@uic.edu - ORCID: 0009-0007-9016-2431

I. Introduction

Research in the machine learning field has found the problem of how to effectively deploy Convolutional Neural Networks (CNNs) on edge devices to be an important question. The trouble of putting up robust models on microcontrollers like the ESP32 WROOM could hardly be overstated, and these limitations emerge from hardware constraints of a very small scale. The research specifying hardware details revealed that the remote server managing the ESP32 has 4 MB of flash memory available, and the most limiting factor is quite evidently the DRAM size of approximately 312 KB as it must be shared with the subroutines of the operating system. Standard architectures, such as Resnet-18, were first considered, and the size of the parameters' size (approximately 46.8 MB at FP32 precision) was pointed out to be the resource that limits the direct deployment of these models by several magnitudes. Hence, the present work is focused on the problem and aims to achieve the parameter reduction of a model through an aggressive compression pipeline that shall still enable classification of the data with an acceptable level of accuracy. Several optimization methods are incorporated into our approach. We achieve structured pruning based on an importance measure by cutting out entire convolution layers by eliminating those whose weights have the lowest magnitude. Memory requirements of dense layers are met by the introduction of Adaptive Average Pooling (AAP), which leads to a significant reduction in the size of the input feature map. In addition, an 8 bit post training quantization is performed with the idea of storing the data in 4 times less space than 32 bit floating point precision. To speed up the inference further, the changes on the algorithmic side are supported by low level C optimizations on the ESP32 hardware, such as output stationary dataflow and convolutional boundary checking.

A. Contributions

Alp Berke Ardic decided on the neural network model to be used in the project and determined the compression algorithms. He wrote the Python files for training and fine tuning the NN model, generating the header files for weights, biases, and model specifications for ESP-32 deployment along with binary files that Dominik Kornak used for deploying the model on ESP32. Moreover, Alp provided the CPU metrics for one-shot inference for a baseline against ESP32. He also provided the model size vs. accuracy under different compression algorithms, demonstrating the compression algorithm's efficient design. He wrote Section II to demonstrate the reasoning and methodology behind the compression methods and showcase the results.

Dominik Kornak deployed the trained Neural Network onto the ESP32, optimized inference and accumulated all inference time, accuracy, and memory metrics for the ESP32. Developed all the embedded C code for ESP32 in the ESP-IDF framework, prioritizing memory usage and accuracy, and optimizing all layers in the network, focusing heavily on convolution. He then ran all inference on ESP32 with the C code. Also developed Python code that converts a JPG image to .bin file. He also ran power simulation on ESP32 and exported data to Damien Karpen. Developed and wrote Sections III to V, and Section IX.

Damien Karpen was responsible for the power analysis of the deployed CNN on the ESP32. Working with raw measurement data captured by Dominik using the FNIRSI FNB58 USB power tester, he developed a complete analysis pipeline. This consisted of a C# file converter and a Python-based signal processing script to detect and quantify individual inference events. He implemented band-pass filtering to automatically isolate inference operations, enabling precise characterization of power and energy per inference. In addition, he generated visualization plots to validate the energy efficiency of the compressed CNN model. Developed and wrote the material in VII.

Anand Pudi used CACTI 7.0 to simulate a 128 KB SRAM scratchpad based on 45 nm process technology to assess memory energy consumption. Installed and built CACTI on his desktop terminal and also created a configuration file for the ESP32-like SRAM to output the simulation data. This data is

based on the inference code. He also derived the total dynamic energy for the convolutional layers and made comparisons between the 2 layers. Developed and wrote Sections I, VI, and VIII.

II. Pre-Deployment Phase

A. CNN Model Selection Before ESP32 Deployment

Deploying a CNN on an ESP32-WROOM has been challenging due to the memory constraints of its flash memory being 4 MB and DRAM being 312 KB, which led us to implement an aggressive compression algorithm that minimizes the number of parameters of the model while maintaining an acceptable model performance at the same time. Despite flash memory being 4 MB, it is both occupied by ESP32's own memory and putting the model weights into flash memory and retrieving these weights from there at every time we need them would take a long time, this 4 MB memory limitation is not for our actual limitation. Our actual limitation has been determined by DRAM whose size is 320392 bytes (around 312.10 KB), which cannot be utilized in full all the time as ESP32's own subroutines also use it. In order to achieve a compressed model that does not give away its model performance while still being deployable to the ESP32, we initially thought about going with larger models like ResNet-18 that work well with image datasets like CIFAR-10 and compressing them into smaller models. However, as ResNet-18 has around 11.69 million parameters [1], and considering FP32 precision initially (4 bytes each parameter), the model would end up being 46.8 MBs that would need orders of magnitude compression, which would be both waste of training and resources. Thus, we initially started with a smaller CNN with the following specifications:

- **Conv1:** 3 input channels, 32 filters 3×3 kernel, stride 1, padding 1.
- **Maxpool1**: 2×2
- **Conv2**: 32 input channels, 32 output channels, 3×3 kernel, stride 1, padding 1
- **Maxpool2**: 2×2
- **Flatten**: Converts $8 \times 8 \times 32 = 2048$ features
- **FC1**: 2048 inputs, 32 outputs
- **FC2**: 32 inputs, 10 outputs (number of classes)

After the compression methods to be explained in the later subsections, it turned out that the model accuracy was not enough to be considered successful enough (it was around 64.2% accuracy with some deviation). Therefore, we widened our model and also added Adaptive Average Pooling to our model that will also be explained in the compression subsection. The latest model we have chosen is the following model:

- **Conv1:** 3 input channels, 32 filters 3×3 kernel, stride 1, padding 1.
- **Maxpool1**: 2×2
- **Conv2:** 32 input channels, 64 output channels, 3×3 kernel, stride 1, padding 1
- **Maxpool2**: 2×2
- **AdaptiveAvgPool2d:** Output size 2×2
- **FC1**: $64 \times 2 \times 2 = 256$ inputs, 128 outputs
- **FC2**: 128 inputs, 10 outputs (number of classes)

With this model that is slightly wider in the CNN layers and less dense fully connected layer whose reasoning is also to be explained in the compression section, we were able to achieve 72.63% accuracy. The simplicity of our model makes it compatible with our custom ESP32 C inference kernel.

B. Importance-Based Structured Pruning for Model Reduction

For the first compression method we utilized in this project, we have used importance-based structured pruning, where entire convolution layers are removed instead of individual weights depending on the importance of these layers. For each convolution layer, we computed the importance score which is computed by the sum of absolute values of all weights belonging to that unit as the following formula:

$$s_c^{(1)} = \sum_{i,j,u,v} \left| W_{c,i,u,v}^{(1)} \right|, \qquad c = 1, \ldots, C_1,$$

and for the second convolutional layer, it is

$$s_c^{(2)} = \sum_{i,j,u,v} \left| W_{c,i,u,v}^{(2)} \right|, \qquad c = 1, \ldots, C_2.$$

Channels and neurons with higher weight magnitudes are considered as more important while the ones with lower magnitudes are considered less important. The hyperparameter that can be utilized for this selection is named as "pruning ratio". For both of the convolutional layers, we have chosen a pruning ratio of 20%, which means that we kept the top 80% of the channels with the highest importance while getting rid of 20%. Once the top 80% channels are identified after the training, a smaller version of the network containing the surviving channels and neurons is rebuilt and the corresponding weights are integrated into this new architecture. This method created a smaller CNN that is more efficient to run on ESP32 while preserving the model's accuracy well. In the initial stage of the ESP32 deployment, the pruning ratio was selected as 40% so that we could prune it even more. However, after realizing that most of the memory requirements are due to the fully connected layers, we came to the conclusion that the reduction in fully connected layer weights is more important than the reduction in the size of convolutional layers in terms of memory. For that reason, initially fully connected layers were also pruned as the first solution; however, it was not enough, and we decided to use Adaptive Average Pooling that significantly reduced the size of the fully connected layers into ESP32's memory.

C. Adaptive Average Pooling for FC Layer Reduction

As discussed above, we used a 2×2 Adaptive Average Pooling (AAP) layer that forces the spatial resolution of the feature map to a fixed size. After the second convolution and max-pooling, the feature map is $64 \times 8 \times 8$ disregarding the pruning. The way the AAP layer works is by compressing that map to $64 \times 2 \times 2$ by dividing each channel into four regions

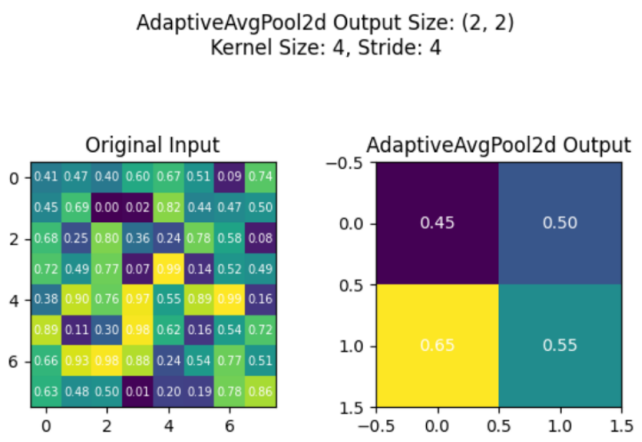


Fig. 1: Adaptive Average Pooling Example with Output Size (2,2) [2]

and averaging them, which serves as spatial downsampling that is explained in the following formulation:

$$\text{AAP} : \mathbb{R}^{64\times 8\times 8} \to \mathbb{R}^{64\times 2\times 2}$$

An example of AAP can be seen in Figure 1 [2].

Using the AAP formulation defined above, the input to the fully connected layer becomes only 256 values, which is 16 times smaller than before (meaning that the reduction in the first fully connected layer is 93.75%). Due to this huge decrease in the number of weights, we removed the fully connected layer pruning, as its benefits would not be too much compared to the AAP. When the AAP and pruning was not in use, our original model's number of parameters was 545098, whereas the AAP and pruning reduced 28576 parameters, which is a compression rate of 94.84% in terms of number of parameters. Most of this reduction came from the reduction in the first fully connected layer's input size, while the remaining came from the pruned layers. At this point, as the compression rate was large and we could fit our model into ESP32, we decided not to implement another compression method but rather stick with our current model and keep its performance metrics useful.

D. Fine Tuning After Pruning

Due to the fact that pruning removes some channels in our neural network model, with the newly built model, the accuracy dropped significantly. When we checked the accuracy of the non-pruned, non-quantized model, we reached an accuracy of 76.03%. After the pruning, the accuracy has dropped to 44.84%, which is far away from what is acceptable for a neural network model. To recover the performance of the model, we have applied a short fine-tuning stage that allows remaining parameters to adapt to this new architecture. During this stage, we have trained the model for additional epochs (10 epochs) with ten times smaller learning rate so that the remaining weights do not shift too aggressively and rather recover accuracy in a more smooth way without getting rid of what has already been learned in the initial training phase. Using the fine-tuning phase allowed us to have an accuracy of 73.12%, which is 28.28% increase compared to the pre-fine-tuning accuracy and is comparable to the initial model accuracy.

E. Post-Training 8-bit Quantization

After the fine-tuning phase, we have implemented 8-bit quantization. Since the unquantized FP32 model cannot fit into ESP32 efficiently without a need to use Flash memory, there was a need to quantize our weights into 8 bit integers, which also provides four times reduction in storage size when we wanted to flash our weights into ESP32 as well as upload them into DRAM.

We use symmetric per-tensor linear quantization. In this methodology, each weight tensor is scaled by the tensor's maximum absolute value, and the scaled weights are obtained by rounding these scaled values to the nearest integer. For the formulation of this methodology, let x denote a FP32 weight and let b be the number of bits (here $b = 8$). We first compute the maximum representable integer value as

$$q_{\max} = 2^{b-1} - 1 = 127,$$

and estimate a scaling factor from the maximum absolute weight value:

$$\text{scale} = \frac{\max |x|}{q_{\max}} = \frac{\max |x|}{127}.$$

Each real-valued weight x is then mapped to an integer value q with

$$q = \text{round}\Big(\frac{x}{\text{scale}}\Big),$$

followed by limiting to the valid signed range,

$$q \in \big[-128,\, 127\big].$$

While doing inference, a real-valued approximation of the weight can be recovered by dequantization as

$$\hat{x} = \text{scale} \cdot q.$$

In our implementation, we store the integer tensor q (as `INT8`) together with its corresponding floating-point scale parameter.

F. Comparisons Between Compressed and Non-Compressed Models on CPU

The Table I demonstrates the comparisons between the sizes of different levels of compressions and quantizations along with their accuracies. From that table, it can be seen that the latest version of our model is quite lightweight and comparably similar in terms of the accuracy performance while being comparably faster in a laptop's CPU, as well as being deployable to the ESP32.

Figure 2 demonstrate the trend for size vs. accuracy for different model compressions, and is supplied as a histogram derived from the Table I.

Moreover, to be able to see how our model that is deployed in ESP32 is working, we have also analyzed the single-inference time, energy per inference, estimated CPU Power (W) per inference in the Table II using our latest model.

TABLE I: Model Size and Accuracy under Different Quantization, Pooling, and Pruning Schemes

Model and Quantization	Size (MB)	Accuracy
Original (FP32)	2.204	76.03%
Original (INT8)	0.551	75.12%
Avg. Pooling w/o Pruning (FP32)	0.329	73.96%
Avg. Pooling w/o Pruning (INT8)	0.082	73.05%
Avg. Pooling with Pruning (FP32)	0.207	73.12%
Avg. Pooling with Pruning (INT8)	0.052	72.63%

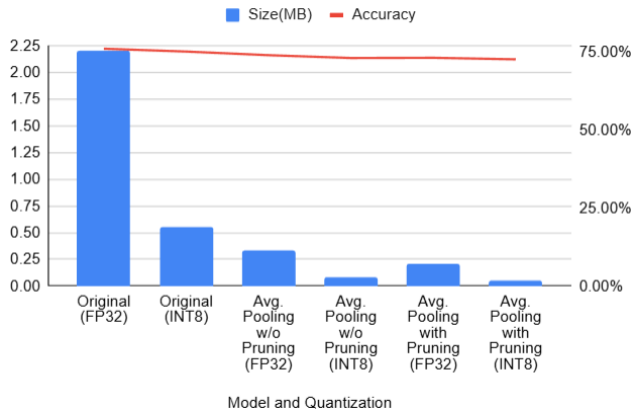


Fig. 2: Statistical Summary of Inference Measurements(CPU), adapted from Table I.

For each of these measurements, a single inference was made, and the metrics were drawn from the CPU directly. In order to have more accurate and understandable data, this has been run multiple times and the average and standard deviations are noted in the Table II. To get these metrics, dequantization has been conducted as described in II-E first, and then the model made an inference using a single image. While the goal of this project was not to deploy it on CPU, we believe that these metrics can serve as a baseline for further comparisons. It is also important to note that CPU does not stop while the inference is being made, but rather does other background work, as well.

III. Input and Weight Implementation on ESP32

To achieve the greatest accuracy, we compute inference in FLOAT32. This does lead to longer inference latency; however, our goal is to compress storage as much as possible and achieve the greatest accuracy we can. Since we store input and weights/biases as INT8 in flash and load them to RAM, we use minimal RAM with this technique (31KB for the image and weights/biases). First, we convert the input image before calling the first convolution function. We take each value from the imagedata array stored in RAM and first convert the raw INT8 value to a FLOAT32 temporary value ranging from [0,1] with

```
float x = (image_data[i] + 128) / 255.0f;
```

Secondly, we normalize this temporary value so its average is 0, ranging from roughly (-2,2). We subtract the mean of the respective channel from the given temporary float value and divide it by the respective standard deviation to compute the normalized float 32 input image. We get these mean and standard deviation values from the CIFAR-10 dataset, which there are three of; one mean and standard deviation per input channel. We normalize the input image with

```
x = (x - mean[c]) / stdv[c];
```

This normalization is compatible directly with the scaled FLOAT32 weights discussed in the next section.

Similarly to the input image, the weights have to be converted from INT8 to FLOAT32 for inference to be done. The conversion is done only when the weights and biases are used, i.e., the convolution and fully connected layers. These biases and weights are converted using their respective scale factor which was derived in section Section II-E. For convolution, biases are converted per output channel when they are preprocessed, and weights are converted to FLOAT32 per kernel. For the FC layer, both biases and weights are converted once per output element. This minimizes the use of temporary buffers, which reduces RAM allocation.

TABLE II: Statistical Summary of Inference Measurements (CPU)

Metric	Value
One-shot Inference Time	0.419 ± 0.0801 ms
Energy per Inference (One-shot)	340.927 ± 37.10 μJ
Estimated CPU Power (One-shot)	0.700 ± 0.0735 W

A. Input and Weight Compression on ESP32

The input for the first convolutional layer or conv1, is a 32x32 image with 3 channels. Each channel represents a color from R,G,B (Red,Green,Blue), giving us a total of 3,072 input pixels. Each of these pixels are a signed INT8 number ranging from [-128,127]. This is achieved by taking a .jpg image from any of the 10 classes in CIFAR-10 class set, using the PIL (Python Image Library) and resizing every image to a $32 \times 32 \times 3$.

When weights or input are used with C code, their data is written to a C header file (.h) and included in the main program file for use. For example, TensorFlow for Microcontrollers [3] exports its weights and biases as a static C array to a *.h* file. This has unnecessary overhead information that takes up a large percent of the files size. A better approach to storing network parameters and inputs is using a binary (*.bin*) file that stores all the information consecutively. In our network, an *.h* file containing weights and biases would be 96 KB of data. Using a *.bin* file, we store the same weights with only 28 KB of data. We save memory size by $3.43\times$, and we retain the same exact data information. For example, storing 20 INT8 weights in a binary file only takes 20 bytes of data. Since one INT8 weight is 1 byte (8 bits), the first byte of data in the file is the first weight. Combining all of these sequentially, we have a 20 byte string of weights. We know the sizes of each bias and weight array, which we pre-define in our code, and use

these as offset indicies throughout our *.bin* file. So if the bin files have contents of [convbias1, convweight1, convbias2,...] and we know the size of every bias/weight, we can access any portion of data we choose, just like a *.h* file with only computing a simple index addition. We implement the same method on our input image, which reduces each image from 11KB of INT8 values to 3 KB of INT8 values in a *.bin* file. That's a $3.647\times$ memory decrease in storage. We then store these *.bin* files into the flash (main) memory and then load these arrays into RAM for inference.

B. Convolution Optimization on ESP32

Convolution being the most computational demanding and energy hungry layer in neural networks, optimization of it is key to minimize latency and power consumption. Since our convolution is designed using output stationary (OS) dataflow, it minimizes the amount of buffers used and keeps a streamlined process in computing output. In an output stationary dataflow, each output element is kept stationary in a buffer while its corresponding input activations and weights are streamed through to compute the final output pixel.

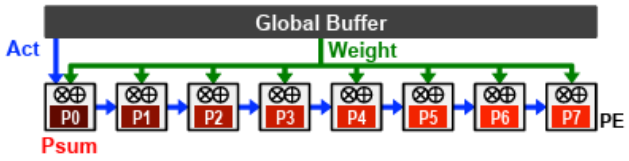


Fig. 3: Output-stationary dataflow diagram, adapted from [4].

Since we need buffers (arrays) for temporary outputs, we create one buffer per output layer to store the temporary outputs of the current layer. This minimizes buffer use to just one FLOAT32 output buffer per layer. Other dataflows, such as row stationary dataflow, would require a buffer that stores the outputs of the convolution and also two additional buffers, one that stores the given weight row and one that stores the given input row, increasing the memory overhead. Currently, the biggest temporary buffer we have is the conv1 output buffer which is 100KB in size. This makes up approximately 32% of the total DRAM storage. Taking into account all temporary buffer sizes (total of 152.14KB), they make up 48.75% of the total RAM capacity. For reference, all the network weights and input image only make up 9.93% of total DRAM storage. So the biggest bottleneck in memory storage is not the actual network parameters its the temporary buffers that are need to store intermediate outputs for each network layer. Using Table III, approximately 85.4% of runtime(Initial-Remaining) memory is occupied by temporary buffers. In table III, the DRAM/IRAM have a shared region of 128KB of memory, which is why total memory does not sum up to 512KB. If you factor this shared memory in with 64KB of cache (32KB per core), you get around 493KB of memory which is an estimate and close to the total RAM capacity. So with our current inference, we have 138 KB of IRAM/DRAM remaining.

We also implement bias preprocessing, which sets every output element to its respective FLOAT32 bias value before

TABLE III: ESP32 Memory Summary (Initial vs Remaining)

Memory Type	Total Memory (KB)	Remaining Memory (KB)
DRAM	312.10	133.92
IRAM	372.97	196.59

inference. This is done to minimize computation in inference since every output element needs a bias addition. Once inference starts and we accumulate the contributions from input channels and weights, we add these partial sums onto the preloaded bias, resulting in the final output element value.

Another optimization we implement is unnecessary bound checking which is a padding alternative. In our convolution, we use a $3 \times 3 \times 3$ filter and convolve that across the input, which produces one output pixel. Since we have to check if our kernel fits entirely inside the input feature map (meaning it is not out of bounds), we have to check the dimensions of the input. However, the kernel will never be out of bounds unless it is touching the perimeter of the input array. If the kernel is within the image and not touching the boundary, we can perform convolution without having an if statement that unnecessarily does a boundary check every time. We accomplish this by looping through the H and W (Height and Width) of the input and starting off at index 1 instead of 0 and ending at the second to last index or (H - 1, W - 1). Since the first and last index of H and W make up the perimeter of the input array, we avoid unnecessary boundary checking for most of the input image. For example, our 32×32 input has 1024 pixels, and taking away the perimeter makes it 30×30 or 900 pixels. This means that for 900 pixels, we do not perform a boundary check, which is 87.9 percent of pixels. The other 124 perimeter pixels require boundary checking during convolution, which takes more time to compute. By avoiding these unnecessary boundary checks, we reduce the number of conditional statements, which improves the overall performance of convolution while mimicking padding.

C. Other Optimizations on ESP32

Traditionally, we implement ReLU after every convolutional layer to introduce non-linearity. This means after every conv call, we have to call a ReLU function, take its output, and then feed it into a pooling function, which involves more computational steps. In our implementation, we fuse the ReLU function into max pooling to avoid the extra function calls. In a 2×2 max-pooling function, you access a 2×2 section of the input, find the maximum value of this input, and have that one maximum value represent those 4 values in a compressed array. We introduce ReLU by applying it to the 4 loaded values before the max is applied. ReLU terminates any negative values, so the inputs to the max function call can only be 0 or a positive value. This method avoids additional ReLU function calls for convolution, which decreases inference time.

Additionally, another small optimization we implemented was storing functions into IRAM. IRAM, or Instruction Random Access Memory, is a type of memory that stores instructions on-chip closer to the CPU, which leads to faster

direct access of function calls. To define this in C code, you label a function as such

```
void IRAM_ATTR app_main(void)
```

Finally, we implement loop unrolling, which, instead of incrementing in a loop by 1 and performing a single task for each iteration, it increments by a predetermined size (x), that won't have pipeline stalls, and performs x amount of tasks per cycle, improving the performance of operation. We use this technique on both the fully connected and ReLU functions, where loop unrolling is more applicable.

IV. Optimization Performance on ESP32

Overall, we achieve around a 313.33 ms inference time with the CIFAR-10 dataset trained on a tiny CNN of 2 convolutional layers and 2 fully connected layers in float 32 precision. Without IRAM and Loop Unrolling, we achieve an inference time of 314.00 ms, which is a .7 ms difference, which is a .22%. It is a very minimal optimization; however, it is easy to implement and always makes a more efficient inference.

As stated before, convolution is the most computationally demanding layer in neural networks. For the ESP32, Conv1 takes up 55.1 ms of inference time, and Conv2 takes up 245.98 ms. This means that the convolution on the ESP32 takes up 96.08% of the total inference time. That's why optimization of convolution is a key factor for all neural network implementations today.

V. ESP32 Inference Results

We ran inference on ESP32 for 10 randomly selected JPG images chosen from Google Images (1 random image per class). The ESP32 produced the correct inference for 9/10 images, demonstrating proper implementation of our trained network. The image misclassified was "Bird" with our network predicting deer with an output class of (2.8040) and bird coming in at second with (1.4159). This is expected as we ran inference on the same model on CPU with more data, and calculated the accuracy of our network was around 72.63%. The CIFAR-10 prediction results on the ESP32 are located in Appendix C.

VI. ESP32 CNN Inference Memory Simulation

In order to assess the memory subsystem energy consumption of the CNN inference on the ESP32, we resorted to using the CACTI 7.0 simulation tool to simulate a 128 KB SRAM scratchpad based on 45 nm process technology. It showed a block size of 64 bytes (which is the same as 16 single precision floats) with a dynamic read energy of 0.0919 nJ and a write energy of 0.1815 nJ per access. Matching these hardware specs with the logical memory access patterns of the C implementation, we got the total dynamic energy for the convolutional layers. The first convolutional layer (Conv1) processing a 32×32 input with 3 channels was estimated to require about 43,200 physical read accesses for inputs and weights, thus the total energy consumed was roughly 8.23μJ. Although the second layer (Conv2) operated at a smaller 16×16 spatial resolution, it still caused a much higher energy cost due to the increased channel depth (25 input channels to 51 output filters). Conv2 was estimated to require around 183,600 physical read accesses, the total energy consumption being 33.89μJ. This comparison highlights that channel depth is the main factor that influences memory energy consumption in this network architecture, that is, the spatial resolution, with Conv2 consuming roughly $4.1\times$ more energy than Conv1. This makes sense because Conv2 takes about $4.46\times$ longer to complete than Conv1. The memory simulation results are located in Appendix A and Appendix B.

VII. Power Consumption

A. Measurement Methodology

To determine the power consumption of our CNN model on the ESP32, we used the FNIRSI FNB58 USB power tester to capture high resolution power measurements during inference operations. We used it to capture voltage, current, the D+ and D- rails, and power at a its maximum rate of 100 samples per second.

To process the data created by the FNB58, we needed two separate steps. The first was to take the binary CFN files captured by the tester and convert them into a CSV format using a C# converter that ran on the CFNReader library. This converted the files into timestamped voltage (VBus), current (IBus), and instantaneous power measurements. The second step was to take the new CSV files and run them through a python-based signal processing program. This program would detect and quantify individual inference events using a bandpass filtering technique with automated threshold detection.

B. Inference Detection & Energy Calculations

Isolating the individual inference operations from normal background activity required the use of a bandpass current threshold approach. The ESP32 had a baseline idle current of $I_{idle} = 36.4 \pm 0.1$mA when the model was loaded but not actively performing inference. During inference operations, the current consumption would rise between $I_{min} = 42.5$mA and $I_{max} = 43.5$mA, in most cases averaging around 43.0mA.

Our analysis applied several criteria to identify valid inference events. We defined an active inference state as:

$$Active(t) = \begin{cases} 1 & I_{min} < I(t) < I_{max}, t > t_{boot}, \tau > \tau_{min} \\ 0 & \text{otherwise} \end{cases}$$

Where $t_{boot} = 0.8$s represents a boot stabilization period, and $\tau_{min} = 0.1$s is a minimum event duration threshold to reject transient noise artifacts. Continuous active regions are identified by computing the first difference of the active state function and integrating to assign block identifiers. For each detected inference event, the total energy consumption was calculated using:

$$E_{inference} = \sum_{t=t_{start}}^{t_{end}} P(t) \cdot \Delta t$$

Where $P(t)$ is the instantaneous power in milliwatts and $\Delta t = 10$ms.

C. Performance Results & Analysis

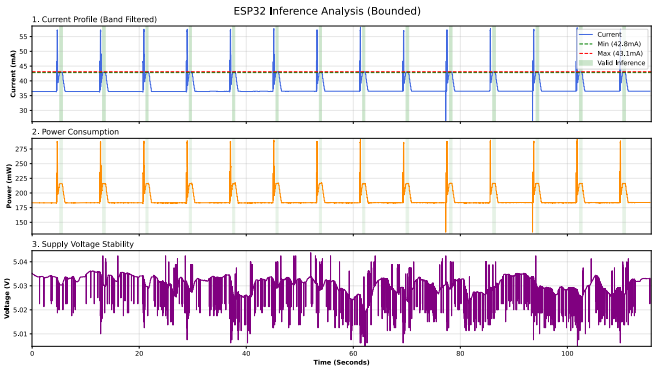


Fig. 4: Power data for the 120s period with 14 inference events.

An analysis of our power measurement data revealed consistent inference behaviour across multiple test runs. We measured 14 distinct inference events over a 120-second monitoring period, showing repeatable energy characteristics with a low inter-event variability. The average current during inference was stable between 42.7mA and 43mA.

The supply voltage remained around $V_{Bus} = 5.030$V with variations within 0.01V during steady-state inference, though there was the occasional, brief spike of 10-12mV, which was likely due to current transients. The average power during inference was approximately 216.3mW, with energy being between 86.34mJ and 108.17mJ depending on the event duration. The total energy consumption of an inference, including the ESP32 boot phase and any data transfer, was approximately 331.9mJ.

D. Power Efficiency & Idle-Active Transition

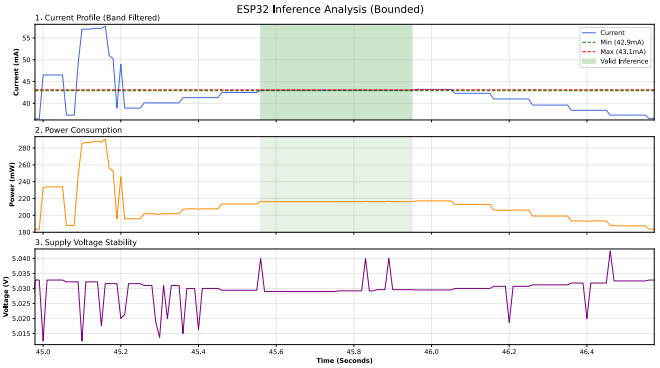


Fig. 5: Zoomed in power data for the inference at 45.5s.

The idle state (model loaded, no inference) consumes approximately $P_{idle} = V_{Bus} \cdot I_{idle} = 5.03\text{V} \cdot 36.4\text{mA} = 183.1\text{mW}$ baseline power. Since power consumption increases to $P_{active} = 216.3$mW during inference, the difference can be calculated by:

$$\Delta P = P_{active} - P_{idle} = 33.2\text{mW}$$

$$\frac{\Delta P}{P_{idle}} = 18.13\%$$

The small delta between idle and active power demonstrates the efficiency of our compression approach. The modest increase in power suggests that the ESP32's idle power is dominated by peripheral operation, clock tree distribution, and leakage current, while our optimized inference adds minimal dynamic power.

VIII. Conclusion

We demonstrated the successful deployment of a compressed custom CNN on the ESP32 WROOM for image classification in this work. We achieved a parameter reduction rate of 94.84% relative to the original uncompressed model by means of structured pruning and AAP. Even with such an aggressive compression, the model obtained a classification accuracy of 72.63% on the CIFAR 10 dataset, and the hardware validation correctly identified 9 out of 10 test images. Performance analysis showed that the average inference time was close to 313.33 ms per image. Our memory analysis revealed that the main limitation for inference on the ESP32, which is not the storage of network parameters, but the allocation of temporary buffers for immediate outputs, which took up about 85.4% of the runtime memory. In addition, we found that convolution operations take up 96.08% of the total inference time, thus emphasizing the critical importance of optimizing convolutional kernels for specific hardware architectures. Energy simulations also revealed that channel depth is the main factor affecting memory energy consumption and the second convolutional layer consumes about 4 times more energy than the first due to the increased number of filters.

IX. Future Work

Since our biggest bottleneck in inference on the ESP32 was convolution, we plan on targeting this area. Currently we use one core on the ESP32 (core0), which is the designated computational core with core1 being designated for Bluetooth and WiFi operations. We can parallelize our convolution code to both of these cores which would in theory half our inference time for convolution bringing our total inference time to around 160 ms with proper implementation. Another potential optimization would be to replace FLOAT32 inference with int8 inference. This would speed up our inference time greatly however it is unknown how much our accuracy would decrease on the ESP32. Since we prioritize RAM allocation and inference accuracy, this could potentially avoid our objective. Also, minimizing the use of temporary buffers could decrease our RAM usage significantly if we could reuse these buffers or find a different technique to store temporary outputs, our RAM usage would decrease significantly allowing the application of bigger and more dense networks with more parameters.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," arXiv preprint arXiv:1512.03385, 2015.

[2] A. Singh, "Demystifying Pooling Layers in CNNs: MaxPool, AvgPool, and More," *ML Made Simple*, 2023. [Online]. Available: https://mlmadesimple.in/demystifying-pooling-layers-in-cnns-maxpool-avgpool-and-more/

[3] "Songspire – Running TensorFlow Lite in a Microcontroller Such as Pico," Element14 Community, 2025. [Online]. Available: https://community.element14.com/challenges-projects/design-challenges/pi-fest/b/blog/posts/songspire—running-tensorflow-lite-in-a-microcontroller-such-as-pico. [Accessed: Nov. 29, 2025].

[4] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," Proceedings of the IEEE, 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8114708. [Accessed: Nov. 29, 2025].

Appendix A
ESP32-WROOM Memory Characteristics

TABLE A.1: ESP32 SRAM Memory Characteristics

Parameter	Value
Technology Node	45 nm
Memory Size	128 KB
Block Size	64 Bytes
Access Time	0.73 ns
Dyn. Read Energy	0.0919 nJ
Dyn. Write Energy	0.1815 nJ
Leakage Power	65.41 mW
Area	0.44 mm^2

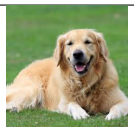
Appendix B
Model Deployment Memory Energy Consumption Results for Convolutional Layers

TABLE A.2: Memory Energy Consumption Results

Layer	Log. Reads	Phys. Access	Energy (μJ)
Conv 1	691,200	43,200	8.23
Conv 2	2,937,600	183,600	33.89

Appendix C
Example CIFAR-10 Predictions on ESP32

TABLE A.3: CIFAR-10 Predictions on ESP32

Class	**Image**	**Inference Time (ms)**	**Top-3 Predictions**
dog		313.33	1. dog (3.55) 2. horse (1.68) 3. cat (0.44)
truck		313.38	1. truck (5.27) 2. automobile (-0.01) 3. airplane (-1.37)
airplane		313.39	1. airplane (4.95) 2. bird (-1.34) 3. cat (-1.36)
automobile		313.36	1. automobile (9.09) 2. truck (4.11) 3. airplane (1.47)
deer		313.38	1. deer (5.60) 2. horse (1.59) 3. airplane (-0.08)
frog		313.35	1. frog (3.39) 2. bird (1.24) 3. cat (0.60)
cat		313.37	1. cat (2.82) 2. frog (1.23) 3. dog (-0.22)
ship		313.37	1. ship (7.90) 2. airplane (1.52) 3. bird (-2.17)
horse		313.34	1. horse (5.38) 2. deer (2.04) 3. bird (1.43)
bird		313.36	1. deer (2.80) 2. bird (1.42) 3. dog (0.02)