A REPORT

ON

OPTIMIZATION OF DEEP LEARNING ALGORITHMS FOR RESOURCE-CONSTRAINED DEVICES

BY

Aditya Jagadish Bhat	2021A7PS2071G
Abhishek Kamat	2021AAPS2932G
Rishabh Mittal	2021A7PS2620H

AT

Central Electronics

Engineering Research

Institute, Pilani

A Practice School-I Station

of

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

(July, 2023)

A REPORT

ON

OPTIMIZATION OF DEEP LEARNING ALGORITHMS FOR RESOURCE-CONSTRAINED DEVICES

BY

Aditya Jagadish Bhat	2021A7PS2071G	Computer Science
Abhishek Kamat	2021AAPS2932G	Electronics & Communication
Rishabh Mittal	2021A7PS2620H	Computer Science

Prepared in fulfillment of the
Practice School-I Course Nos.
BITS C221/BITS C231/BITS C241

AT

Central Electronics Engineering

Research Institute, Pilani

A Practice School-I Station

of

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

(July, 2023)

Acknowledgements

Firstly, we would like to thank Birla Institute of Technology and Science, Pilani, and Central Electronics and Engineering Research Institute (CEERI) for giving us this opportunity to research the topic “Optimization of deep learning algorithms for resource-constrained devices” under the coursework of Practice School I. We want to express profound gratitude to our PS faculty, Dr. Meetha V. Shenoy and Dr. Abhijit Asathi of the Electrical and Electronics Engineering Department of Birla Institute of Science and Technology Pilani, for their contributions to this project. We want to express our special thanks to our mentors, Dr. Sanjay Singh and Dr. Sumeet Saurav, for the time and efforts they provided throughout the project. Their advice and suggestions were immensely beneficial to us during the project. We thank them for instilling us with knowledge throughout all our meetings during the Practice School I course.

Abstract Sheet

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN)
Practice School Division

Station: CEERI **Center:** Pilani

Duration: 8 weeks **Date of Start**: 30 May 2023

Date of Submission: 19 July 2023

Title of the Project: Optimization of deep learning algorithms for resource-constrained devices

ID No./Name(s)/ Discipline(s)/of the student(s)

Aditya Jagadish Bhat	2021A7PS2071G	Computer Science
Abhishek Kamat	2021AAPS2932G	Electronics & Communication
Rishabh Mittal	2021A7PS2620H	Computer Science

Name(s) and designation(s) of the expert(s): Dr. Sanjay Singh (Principal Scientist & Group Head)

Name(s) of the PS Faculty: Dr. Meetha V. Shenoy

Key Words: Microcontrollers, Edge Computing, Deep Learning, TinyML

Project Areas: Deep Learning on Microcontrollers, Machine Learning Optimization

Abstract:

This project presents an approach to deploying a gesture recognition convolutional neural network (CNN) model on the Arduino Nano 33 BLE Sense Lite microcontroller using TensorFlow Lite for Microcontrollers. By fine-tuning a MobileNet-v2 model on a custom dataset of labeled (Rock, Paper, Scissors) gesture images, real-time and accurate gesture recognition is achieved in a resource-constrained environment. This implementation demonstrates the potential for integrating deep learning on edge devices, opening up new opportunities for interactive and portable applications in various domains.

Signature(s) of Student(s) Signature of PS Faculty

Date 19/07/2023 Date

Table of Contents

Introduction

Problem Statement

This project addresses the challenge of real-time and accurate gesture recognition in a resource-constrained environment. It aims to deploy a gesture recognition CNN model on the Arduino Nano 33 BLE Sense Lite microcontroller using TensorFlow Lite for Microcontrollers. By fine-tuning a MobileNet-v2 model on a custom dataset of labeled gesture images (Rock, Paper, Scissors), the project enables efficient and reliable gesture recognition on the edge device. This implementation opens up new possibilities for interactive and portable applications across various domains by integrating deep learning on edge devices.

Introduction to TinyML

Machine learning has witnessed remarkable progress, enabling us to tackle complex problems with unprecedented accuracy and efficiency. Traditional machine learning models heavily rely on the computational power of cloud infrastructure, where vast amounts of data are processed and analyzed. Well-defined state-of-the-art computer architectures like the TPUs from Google and GPUs from companies like Nvidia and AMD are designed for running ML applications. However, as the demand for intelligent applications grows, there is an emerging need to deploy machine learning models closer to the edge, directly on resource-constrained devices. This field, aimed at implementing machine learning on severely resource-constrained systems that sometimes do not even have an OS, is known as TinyML. There are multiple reasons that drive research on TinyML:

Firstly, latency is critical in many real-time applications, such as autonomous vehicles, industrial automation, and Internet of Things (IoT) devices. These applications often require instantaneous decision-making, where transmitting data to the cloud for processing and waiting for a response introduces significant delays. By deploying machine learning models directly on edge devices, TinyML reduces round-trip time and enables faster, more efficient decision-making.

Secondly, privacy and data security have become growing concerns in the age of cloud computing. Transmitting sensitive data to the cloud for processing raises potential risks of unauthorized access, data breaches, and privacy violations. With TinyML, data remains localized on the edge device, minimizing the need for data transmission and reducing the risk of data exposure.

Thirdly, TinyML enables offline capabilities, allowing edge devices to function independently without relying on continuous cloud connectivity. This ability is particularly beneficial in scenarios with limited or intermittent network access.

Furthermore, cost is a crucial factor to consider. Cloud computing involves data transfer and processing fees, which can be significant when dealing with large volumes of data. TinyML reduces reliance on cloud infrastructure, minimizing data transmission and cloud computing resources costs. As per recent studies, in remote monitoring settings, 99% of raw sensor data is discarded, which is an unused wealth of data for machine learning. TinyML can provide a unique solution: summarizing and analyzing data at the edge on low-power embedded devices, TinyML can provide innovative summary statistics that take these previously lost patterns, anomalies, and

advanced analytics into account [1] [2]. Henceforth, we see multiple possible use cases of TinyML in upcoming times, from predicting faults in industrial appliances before failure to smart home appliances to saving animals from poaching to health appliances to face detection on microcontrollers.

Challenges in Implementing TinyML

One of the challenges with TinyML is the trade-off between model complexity and limited computational resources, requiring careful optimization and model design. In many cases, an economical tiered approach, where a small, always-on piece of functionality determines when to pay attention so that higher-powered hardware and software can kick in. That higher-powered edge solution even can serve as a tier to sending the big problems to the cloud for resolution. An excellent example of that is always-on sound detection, which can be used to detect whether a voice has been heard, which is a small enough problem to do without much computing. If a small always-on chip determines it was a voice, it can wake up the CPU in a phone to see if what was said was a “wake word.” If true, then the entire uttered phrase can be sent to the cloud for parsing.

On a precautionary note, one should be aware of hazards that can happen if we are not conscious and responsible enough while using these technologies. Some problems could be false results leading to life-threatening events, privacy breaches, devices being hacked, or unethical biases in prediction[1].

Literature Review

Initially, we began learning about the field of TinyML by completing the three TinyML courses offered by HarvardX on the edX platform[1][2][3]. The three courses were immensely helpful to understand the subject and get a way forward. The corresponding learnings about TinyML and optimizing ML models is discussed in the Main Text Section. These introductory courses also allowed us to understand the TinyML Cookbook [5] which provides a solid foundation to begin with practical research projects in TinyML.

Puranjay Mohan et. al[6] developed a Tiny CNN for medical face mask detection.The hardware components used for edge deployment is the OpenMV Cam H7, housing STMicroelectronics’ STM32H743VI, an ARM CortexM7 based 32-bit microcontroller and a small camera. After experimenting with different architectures and comparing their size and performance, the proposed CNN architecture was found to be the best(compared with a modified version of SqueezeNet). Post-training TensorFlow-Lite’s Full Integer Quantization was used to convert all three models from float-32 precision to Int-8 precision. The proposed model reached the training accuracy of 99.79%, and achieved a testing accuracy of 99.81% and 99.83% for float32 and int8 models respectively. The 1.52 MB float32 model was reduced to 138 KB post-quantization. This shows that quantized image classifier models can be deployed successfully on microcontrollers achieving excellent accuracies.

Banbury et. al[7] focuses on the application of neural architecture search (NAS) techniques to design accurate machine learning models that meet the memory, latency, and energy constraints of MCUs. The authors make an intriguing observation that model latency exhibits a linear relationship with the model's operation count. Leveraging this insight, they employ differentiable

NAS to search for low-memory and low-operation count models. Experimental results demonstrate the effectiveness of the methodology, yielding MicroNet models with state-of-the-art performance on TinyMLperf benchmark tasks. The work contributes to expanding the capabilities of IoT devices through optimized deep learning algorithms for MCUs using NAS, suggesting that NAS can be used to further optimize models for TinyML applications.

Main Text

Dataset Preparation

We used the Arduino Tiny ML machine learning kit for our project which comes with the Arduino Nano 33 BLE Sense, a OV7675 camera module, and a shield to interface these two devices. The Arduino Nano 33 BLE Sense is compatible with the Arduino ecosystem, making it easy for us to access a vast range of libraries and code. The OV7675 camera module is an image sensor module based on the Omnivision OV7675 CMOS image sensor. It is used for capturing still images in various electronic projects, robotics, and other applications where image data is needed. It can capture images in VGA, QVGA, and QQVGA formats and usually communicates with the host microcontroller using a serial communication interface, such as SCCB (Serial Camera Control Bus) or I2C (Inter-Integrated Circuit).

Figure 1: TinyML Kit used [8]

To prepare a custom dataset using the camera module, we developed a Python script to view the images captured from the Arduino Nano 33 BLE Sense on our device screen. The script also helped to store only those images that were deemed suitable for training our model. To run the script, we needed to import the necessary Python packages: NumPy, PySerial, UUID, and PIL (Python Imaging Library). These packages were essential for image processing and establishing communication with Arduino. The script started by initializing a serial communication link between the laptop and the Arduino. This was achieved by creating a Serial object and ensuring that the correct baud rate (i.e., 115200, for our device) and port settings were configured. This setup allowed for seamless data transfer between the devices. Refer to Figure 2 for the code snippet.

```
port = '/dev/cu.usbmodem14201' # cl
baudrate = 115200 #bits per second

# Initialize serial communication
ser = Serial()
ser.port     = port
ser.baudrate = baudrate
ser.open()
ser.reset_input_buffer()
```

Figure 2:For Serial Communication

After capturing an image, the board stored the data in the form of a byte array. To transfer this image data to our device, we initialized a 3D NumPy array on our device. This NumPy array was used to receive and store the image data for further processing or analysis.

Once we were ready to transfer the data, we first read the width and height of the image captured by the board. We then resized the NumPy array accordingly using the numpy.resize() function. To resize the array, we passed the values of width, height, and the number of channels (i.e., 3, as we were capturing RGB images) as arguments to the function. This ensured that the NumPy array was appropriately sized to accommodate the image data. Next, we started reading the data from the serial communication. The data was received as a string, and to convert it to an integer, we used the int() function. The integer value was then stored in the NumPy array at position [y][x][c] using nested loops. These loops iterated over the width, height, and channels of the image to correctly fill the NumPy array with the received pixel values. Refer to Figure 3 for the code snippet.

```
for y in range(0, height):
    for x in range(0, width):
        for c in range(0, num_ch):
            data_str = serial_readline()
            image[y][x][c] = int(data_str)
```

Figure 3: Transferring Pixel data into numpy array

After the data transfer was complete, we utilized the Python Imaging Library (PIL) to create a new PIL Image object from our NumPy array. This enabled us to work with the image using the functions provided by PIL. We proceeded to crop the image to a dimension of 120x120x3 using the crop function. The cropped image was then viewed using the view function, allowing us to inspect the result. If the image was deemed appropriate to be included in our dataset, we saved it

in the appropriate class by assigning it a unique ID using uuid, and we saved it in PNG format. This ensured that each image in the dataset had a unique identifier and was stored in a standard image format for further use. Figure 4 is the code snippet that demonstrates how to save the image in the appropriate class with a unique ID in PNG format.

```
#if it's a good image
key = input("Save image? [y] for YES: ")
if key == 'y':
    str_label = f"Write label or leave it blank to use [{label}]: "
    label_new = input(str_label)
    if label_new != '':
        label = label_new
    unique_id = str(uuid.uuid4()) #prepare a unique id
    filename = label + "_"+ unique_id + ".png"
    image_cropped.save(filename)
    print(f"Image saved as {filename}\n")
```

Figure 4: Saving images with unique labels

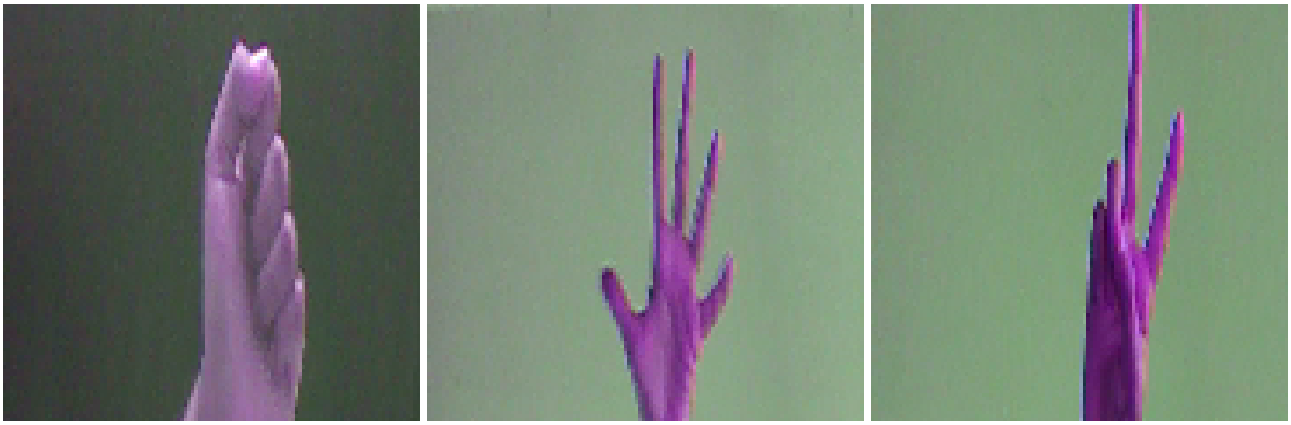

Figure 5:Sample images under the 3 classes (rock, paper, and scissors, respectively)

Following these steps, we prepared a dataset consisting of 150 captured images with dimensions 120x120x3, divided into 3 classes (i.e., rock, paper, and scissors).

Model Development

Image Augmentation

The `image_dataset_from_directory` function in TensorFlow's Keras module provides a convenient and efficient means of constructing a training dataset from image files within a custom dataset. This function plays a crucial role in the preprocessing and organization of image data,

facilitating the training of deep learning models. Upon importing the dataset, a standard practice is to partition it into training and validation subsets. In our project, the training subset comprises approximately 80% of the data, while the remaining 20% is allocated for validation purposes. This partitioning enables model evaluation on unseen data during the training process, effectively assessing its generalization performance.

Preprocessing of the image data begins with resizing the images to a uniform shape of 64x64 pixels with three color channels (64x64x3). The `interpolation` parameter is specified as "bilinear" to ensure high-quality resizing using a method that blends neighboring pixels. This choice of interpolation aims to minimize distortion and maintain the integrity of the image content during the resizing procedure. Bilinear interpolation is also chosen to maintain consistency with image preprocessing done on the Arduino before inference.

To ensure consistent and standardized input for the model, a rescaling layer is applied. This layer normalizes the pixel values, which initially range from [0, 255], to the normalized range of [-1, 1]. This normalization process improves training stability and convergence by reducing the scale of the input data and preventing numerical instability. The normalized range of [-1, 1] is required by the base MobileNet-v2 model since it is pre-trained on images in this range. Both the training and validation datasets undergo the rescaling procedure using the aforementioned layer. This normalization step guarantees that both subsets are treated consistently and eliminates potential discrepancies arising from different scaling factors.

Additionally, a data augmentation pipeline is implemented to augment the training dataset. This pipeline encompasses a series of random transformations applied to the images during training. Random rotation introduces variability by rotating the images by random angles, while random horizontal flipping randomly mirrors the images horizontally. These transformations enhance the dataset's diversity and assist the model in recognizing objects from various orientations. Random zooming modifies the scale of the images, simulating different perspectives or distances. This augmentation technique promotes the model's ability to handle variations in object size and spatial relationships. Furthermore, random translation shifts the images by random distances horizontally and vertically, mimicking potential shifts or displacements in real-world scenarios. This augmentation process enhances the model's robustness to spatial variations and object positioning. It is important to note that the data augmentation pipeline is exclusively applied to the training dataset. By excluding the validation dataset, the evaluation process remains unbiased, as it reflects the model's performance on unaltered data.

Model Architecture

In our project, we employed the MobileNet-V2 model. MobileNet V2, part of the family of pre-trained models offered by Keras, is specifically designed to cater to target devices with limited computational power. Compared to its predecessor, MobileNet V1, MobileNet V2 achieves higher accuracy while using half the number of operations. One of the key design choices that made the MobileNet series suitable for edge inferencing is the adoption of depthwise convolution.

Traditional convolution layers are known to be computationally expensive, especially when dealing with larger kernel sizes such as 3x3 or greater. Additionally, standard convolutional operations

often require extra temporary memory, increasing the computation load. To mitigate these issues, MobileNet V1 replaced standard 2D convolution with depthwise separable convolution. Depthwise separable convolution consists of two main components: a depthwise convolution with a 3x3 filter size, followed by a pointwise convolution with a 1x1 kernel size. The depthwise convolution focuses on channel-wise operations, applying a separate filter to each input channel. This approach significantly reduces the number of trainable parameters, memory usage, and computational cost.

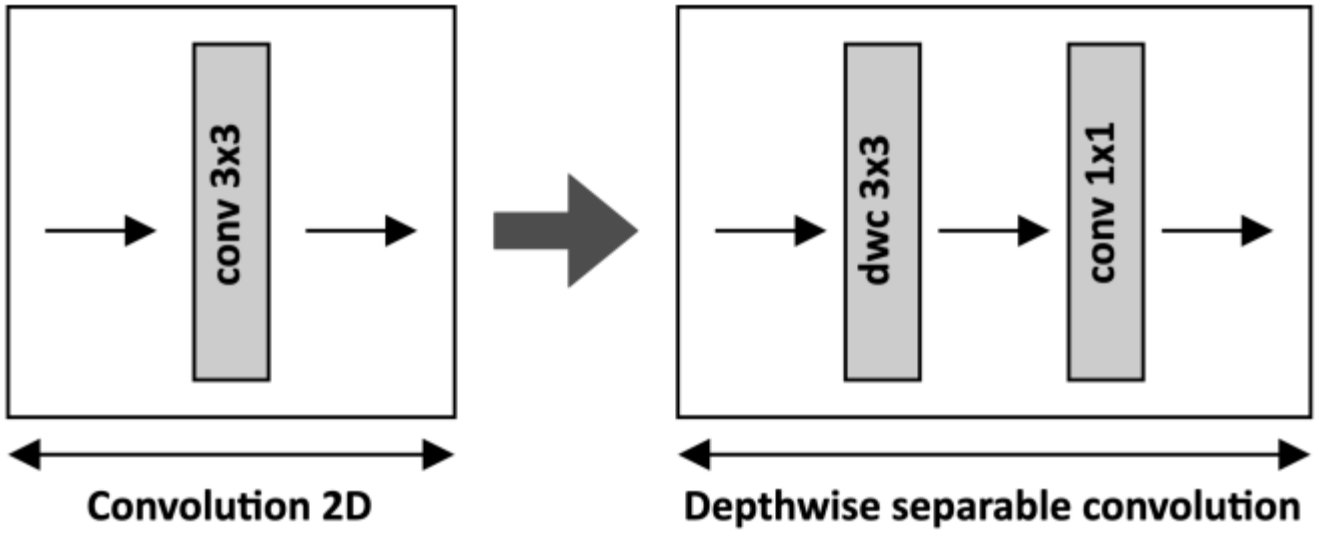

Figure 6: Depthwise separable convolution[5]

With MobileNet V2, the computational cost is further reduced by performing convolutions on tensors with fewer channels. The goal is to have all layers work on tensors with fewer feature maps, which improves model latency. From an accuracy perspective, compact tensors retain the relevant features required to solve the specific problem. However, reducing the number of feature maps using depthwise separable convolution alone can lead to a drop in model accuracy. To address this issue, MobileNet V2 introduces the bottleneck residual block, which acts as a feature compressor. This block ensures that the network retains important information while keeping the number of channels used in the model smaller.

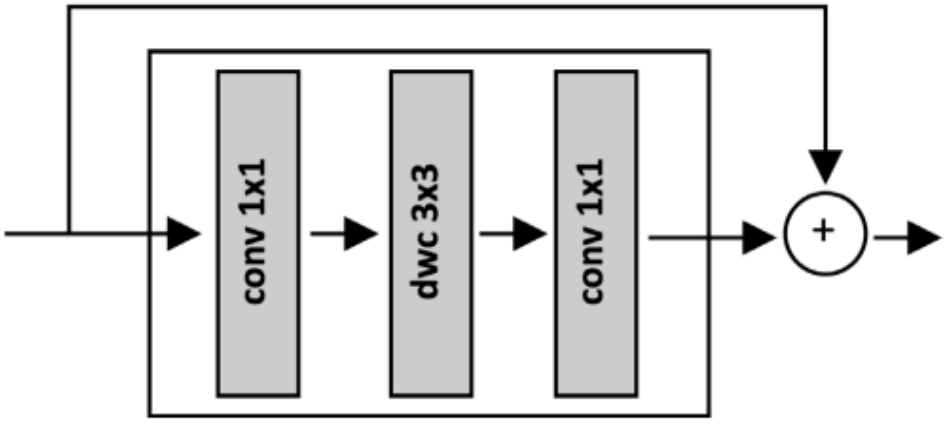

Figure 7: Bottleneck residual block[5]

The bottleneck residual block operates in two stages. Firstly, the input tensor is processed by the pointwise convolution, which increases the number of feature maps. This expansion step allows the model to capture more complex patterns and features. Next, the output of the pointwise convolution is passed to the depthwise separable convolution layer, where the features are compressed into a smaller number of output channels. This compression reduces the model's computational complexity while preserving the critical features.

Fine-tuning is an effective technique in machine learning that allows for training a pre-trained deep neural network using a small dataset, which is particularly useful in cases where collecting a large dataset is challenging or time-consuming. In the context of image classification, traditional training from scratch typically requires a substantial dataset consisting of at least 1,000 images per class. However, fine-tuning offers an alternative approach by leveraging the knowledge captured by a pre-trained model and adapting it to the specific classification task at hand. To fine-tune the model for our image classification task, we made certain modifications. Firstly, we set the include_top parameter to False, which excluded the fully connected layer (top) of the MobileNet-V2 model. By removing this layer, we retained the earlier layers responsible for feature extraction, allowing the model to adapt to the specific characteristics of our target dataset.

To construct a custom model for image classification, we built upon the modified MobileNet-V2 base model. We took the output of the fourth-to-last layer of the MobileNet-V2 base model and introduced additional layers. These included a global average pooling layer, which reduced the spatial dimensions of the feature maps, followed by a dense layer consisting of 64 units. Batch normalization and rectified linear unit (ReLU) activation were applied to enhance the model's learning capabilities. Finally, a dense layer with three units and softmax activation was added for the final classification.In our case, this modification caused the parameter count to be reduced from approximately 619K to 443K, enabling faster inference times and decreased model size without significant loss in accuracy.

The fine-tuning process involved setting the trainable parameter of the base model to True. By doing so, all layers of the MobileNet-V2 model were made trainable. With a low learning rate, this approach struck a balance between leveraging the pre-existing knowledge captured by the model and adapting it to the nuances of our specific classification task. Fine-tuning allowed us to overcome the limitations of limited data availability by leveraging the pre-trained weights and transferring knowledge from the imagenet dataset.

Model Training

The learning rate is a crucial hyperparameter in training deep neural networks. To dynamically adjust the learning rate during training based on a StepDecay schedule, a callback instance called lr_callback is employed. This callback, known as LearningRateScheduler, reduces the learning rate based on the StepDecay schedule after a certain number of epochs by a constant factor, allowing for better convergence and a higher accuracy than would be achievable with a constant learning rate. This scheduler is passed as a callback to the model.fit function, allowing for dynamic adjustments.

The model.compile function configures the model for training. In this case, the Adam optimizer is

used, which is a popular choice for gradient-based optimization algorithms. The loss function employed is sparse categorical cross-entropy, suitable for multi-class classification tasks. During training, accuracy is monitored as one of the metrics. Maximizing accuracy is the primary objective, and by monitoring it, the model's progress can be tracked. The model.fit function is responsible for training the model using the training dataset. By iterating through the dataset multiple times, the model can learn and adjust its parameters to minimize the loss and improve accuracy.

Quantization

Since microcontrollers have very limited compute, memory and storage and since they usually lack an OS, standard neural networks built using TensorFlow can not directly be deployed on a microcontroller. TensorFlow's SavedModel APIs are used to save trained models in TensorFlow. This format allows for easy reusability and compatibility with TensorFlow. The TensorFlow Lite converter is a tool used to convert TensorFlow models into the TensorFlow Lite format, which is optimized for deployment on resource-constrained devices like mobile phones, microcontrollers, and edge devices. The converter converts the model to a format that is more efficient in terms of memory usage and computation[2].

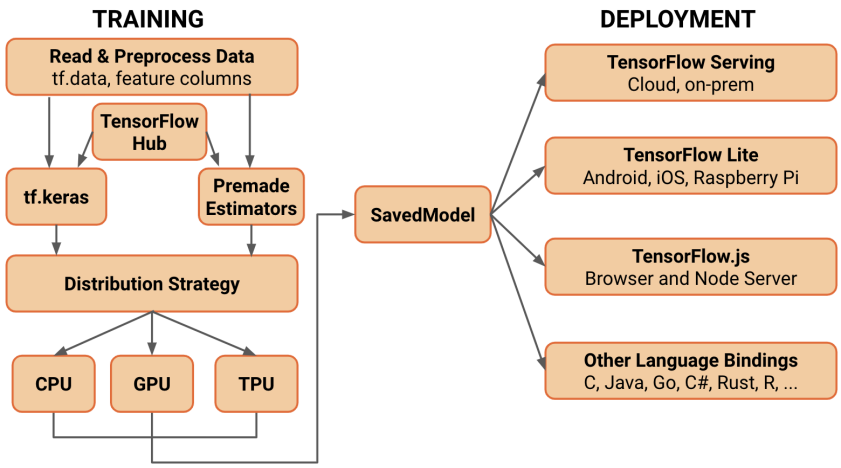

Figure 8: TensorFlow Architecture[4]

When you have a pre-saved .tflite file (a model in TensorFlow Lite format), you can instantiate a tf.lite.Interpreter object in TensorFlow to load and use the model. The 'model_content' property of the interpreter can be set to specify the path or content of the existing model file. Once the model is loaded using the tf.lite.Interpreter, you can perform inference by setting the value of the input tensor, which is typically done by assigning data to the input tensor using the 'set_tensor' method. After setting the input tensor, you invoke the model by calling the 'invoke' method, which leads to a predicted output in the output tensor. Finally, you can obtain the value of the output tensor, which contains the inference results, using the 'get_tensor' method[2].

Quantization refers to reducing the precision of numerical values in a neural network. In a traditional deep learning model, computations are typically performed using 32-bit floating-point numbers (FP32), which provide high precision but require more memory and computational

resources. Quantization involves reducing the precision of these numbers, typically to lower bit-width representations like 8-bit integer (INT8) representations. The reason why this can be done is because weights in a neural network are usually concentrated in a small critical range. This leads to a 75 % decrease in the size of the model and a large increase in the inference speed due to fixed-point arithmetic being faster than floating point arithmetic which uses FPUs that may not be available on all embedded devices, while maintaining an accuracy usually within 1% of the original non-quantized model.

Post-quantization, on the other hand, is performed after the model has been trained using conventional techniques. It involves quantizing the model's weights and activations to lower precision representations as a post-processing step through affine quantization using a scale and a zero point. The equation that governs affine quantization is $x\_q = round(x/S + Z)$ where x_q is the quantized value of x; S is the scale and Z is the zero point. Post-quantization can be applied to an already trained model to reduce its memory footprint and improve its inference speed.

To convert the trained model into the TensorFlow Lite format, the tf.lite.TFLiteConverter is used. This converter takes the saved model and converts it into a format suitable for deployment on resource-constrained devices. To ensure accurate quantization, a representative dataset is provided to the converter as tf.lite.RepresentativeDataset. This dataset captures the statistical properties of the input data, which aids in the per-tensor quantization process. During the conversion process, the converter's optimizations are set to [tf.lite.Optimize.DEFAULT]. These optimizations enable default optimization techniques, such as weight quantization, to reduce the model's size.

The input type for inference is set to tf.int8, indicating that 8-bit integer quantization is utilized for the input tensors, while the output tensors use FP32 outputs by default. To specify the supported operations for the target specification, tf.lite.OpsSet.TFLITE_BUILTINS_INT8 is chosen. This ensures that the converted model supports operations required for 8-bit integer quantization. Finally, the command "xxd -c 60 -i model.tflite > model.cpp" is used to convert the TensorFlow Lite file into a byte array. This transformation allows the model to be embedded into code, facilitating its integration into applications running on microcontrollers.

Model Deployment

Before deployment, it is important to optimize the RAM usage during execution. The input, output, and intermediate tensors of the ML model are just a few examples of the variables allocated during program execution that have an impact on RAM usage. However, the model is not solely responsible for memory utilization. In fact, the image acquired from the OV7675 camera needs to be processed. We do this by fusing crop, resize, rescale, and quantize operators. These operators are used to prepare the model’s input.

Preprocessing on Arduino Nano 33 BLE Sense Lite

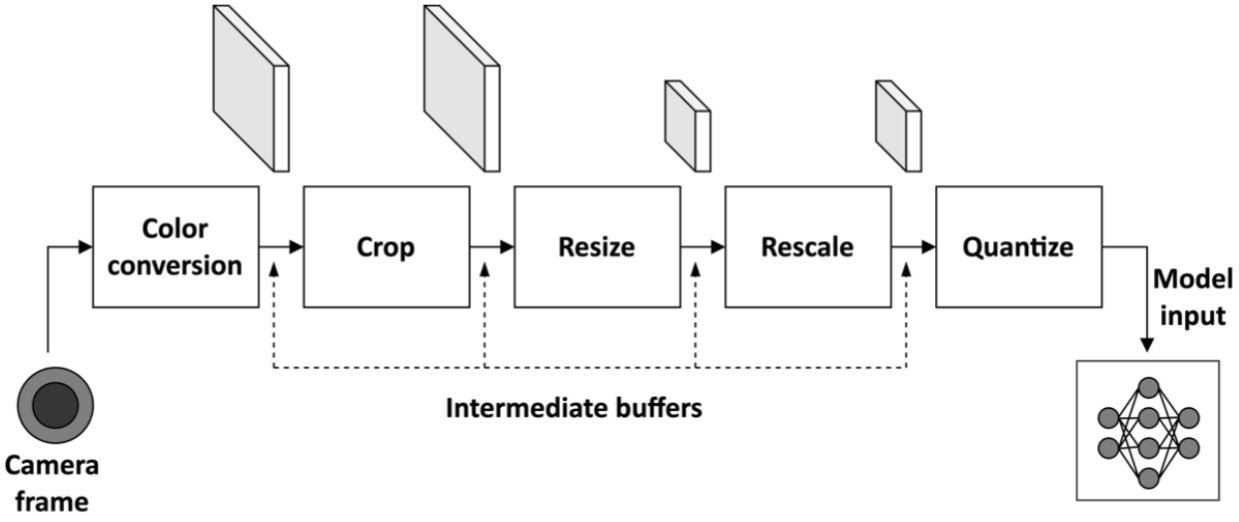

Figure 9: Input preparation pipeline [5]

1. **Color conversion from YCbCr422 to RGB88:**

   The OV7675 is initialized to send QQVGA (160*120) images encoded in YCbCr422 format. YCbCr422 is a color space that separates the luminance (Y) component, representing brightness, from the chrominance (Cb and Cr) components, representing color information. The separation of color information allows for efficient compression, particularly in video applications. However, for image recognition and processing tasks, RGB is a more commonly used color space. It represents color using three channels: red (R), green (G), and blue (B), each with 8 bits of precision. Converting YCbCr422 to RGB888 is essential for image processing tasks that rely on RGB data, such as applying deep learning models that expect RGB input.

2. **Cropping the camera frame from 160*120 to 120*120:**

   Cropping is done to focus on a specific region of interest in an image. Initially, the camera captures images in a higher resolution (QQVGA) of 160 x 120 pixels. However, for the specific image recognition task of rock, paper, scissors gesture recognition, the relevant features are present in a smaller area. Cropping reduces the image size, focusing computation on the important region, and saving memory.

3. **Resizing the image from 120*120 to 64*64 using bilinear interpolation:**

   The image needs to be resized to a smaller resolution (64x64) to match the input size expected by our TinyML model. Bilinear interpolation is used to calculate the values of the new pixels when resizing. It works by averaging the values of the surrounding pixels,

considering their distance to the new pixel location. This interpolation technique results in smoother transitions between pixel values and helps preserve the visual quality of the image during resizing.

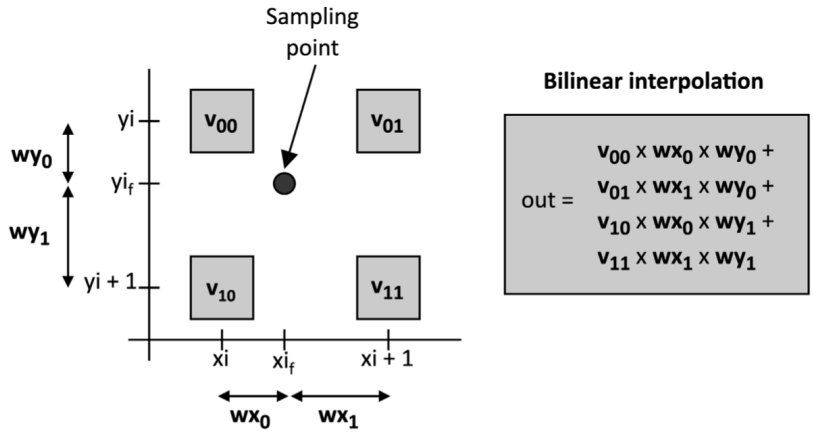

Figure 10: Bilinear Interpolation [5]

4. **Rescaling pixel values from [0,255] to [-1,1]:**

   Rescaling pixel values is crucial for deep learning models that use activation functions sensitive to the input range. For instance, the commonly used ReLU (Rectified Linear Unit) activation function behaves better when its inputs are in a balanced range around zero. By scaling the pixel values from the original [0, 255] range to the normalized range of [-1, 1], the model can handle inputs more effectively. Normalization also helps in dealing with issues like exploding gradients during training and numerical stability during inference.

5. **Quantizing the floating-point values to integers:**

   Quantization maps the continuous range of floating-point values to a finite set of discrete integer values. By converting the floating-point pixel values to 8-bit integers, the memory requirements for storing the input data are significantly reduced, making it feasible to deploy the model on microcontrollers and other edge devices with limited resources. The model's operations can then be optimized to perform integer-only computations, further improving efficiency.

Inference on the Arduino Nano 33 BLE Sense Lite

1. Model Initialization and Memory Allocation: TensorFlow Lite Micro (TFLM) implements sophisticated memory management techniques to handle memory allocation and usage efficiently during model execution on resource-constrained devices like microcontrollers. This is critical to

minimizing memory fragmentation and optimizing the utilization of available RAM. In our code, the TFLM model is initialized by loading the pre-trained model from the rock, paper, scissors C-byte array, which contains the model data. The TFLM interpreter is then created with this model, and a block of memory called TensorArena is allocated to serve as the workspace for storing intermediate results during inference. The size of the TensorArena is set to 160000 bytes, carefully chosen to accommodate the model's memory requirements and any intermediate variables that may be generated during execution. The allocation of a fixed-size memory block ensures that the interpreter has sufficient memory space and minimizes the risk of memory overflow.

2. Micro Interpreter and Ops Resolver: TensorFlow Lite Micro employs a specialized Micro Interpreter designed specifically for running machine learning models on microcontrollers with limited resources. The Micro Interpreter is optimized for memory efficiency, making it suitable for deployment on devices with small RAM capacities. It manages memory allocation and usage during inference, ensuring that the model's operations can be executed within the microcontroller's constraints. The Micro Ops Resolver is another crucial component of TFLM, responsible for mapping the operations present in the TensorFlow Lite model to optimized implementations tailored for microcontrollers. These optimized operations are selected based on the microcontroller's hardware capabilities, allowing for faster execution and reduced memory consumption. By choosing the most efficient set of operations, the Micro Ops Resolver enhances the overall performance of the model on the microcontroller.

3. Model Execution and Output Handling: In our code, the input tensor is used to hold the preprocessed image data captured from the camera. The image is converted to RGB888 format, resized to the model's input size (64x64), and quantized to int8 format to reduce memory usage. These steps ensure that the input data adheres to the requirements of the TinyML model. The TensorFlow Lite interpreter is then invoked to perform the inference process. During inference, the interpreter executes the model's operations using the preprocessed image data and stores the results in the output tensor. In our case, the model is a rock, paper, scissors gesture recognition model with three classes: "paper," "rock," and "scissors." The output tensor contains the probabilities of each class, representing the model's confidence scores for each class. To determine the predicted class, the code iterates through the output tensor, identifies the class with the highest probability (the class with the highest confidence score), and selects it as the final prediction. The predicted class label is then printed to the Serial Monitor, providing real-time feedback about the recognized scene.

By following these steps, our application demonstrates how TensorFlow Lite Micro enables efficient and effective execution of machine learning models on resource-constrained devices, showcasing the power of TinyML in edge computing scenarios. The technical optimizations in TFLM, such as memory management and hardware-specific operations, allow for reliable and low-latency execution of machine learning tasks on microcontrollers.

Challenges Faced

At the initial stages of our project, we encountered challenges with the accuracy of the model predictions, as well as imbalances in the occurrence of different classes within the dataset. These

issues were indicative of poor data quality and highlighted the need for refinement. To address this, we undertook a rigorous process of refining our dataset. This involved carefully reviewing the data samples, identifying and removing any outliers, and ensuring a more balanced representation of each class. By improving the quality and balance of the dataset, we aimed to enhance the model's ability to learn and make accurate predictions.

As part of our troubleshooting efforts, we conducted inferences on our local machine using a local interpreter. Surprisingly, these inferences yielded the expected level of accuracy, which led us to suspect that there might be an error in the deployment code. We diligently reviewed the deployment code, comparing it with the code used for local inference, and identified inconsistencies.

During development, we consulted the TinyML Cookbook as a reference guide. In doing so, we discovered a minute but extremely significant error in the rescaling factor provided in the cookbook. Recognizing the significance of this discrepancy, we promptly corrected the rescaling code, ensuring that the input images were accurately processed and aligned with the pre-trained base model's requirements.

Another aspect we considered was the memory usage of the deployed model. Initially, we set the input image size to the smallest MobileNet value, 48x48 pixels, in an attempt to conserve memory. However, this compromise had an adverse impact on the accuracy of the model. Realizing the trade-off between memory usage and accuracy, we made the decision to increase the input image size to 64x64 pixels. This adjustment enabled the model to capture finer details and improve its overall predictive performance, at the cost of time required for inference and memory.

By refining the dataset, rectifying code errors, and finding the optimal balance between memory usage and accuracy, we significantly improved the performance and reliability of our deployed model. The iterative nature of our troubleshooting process allowed us to learn from these challenges, continuously improve our approach, and ultimately achieve higher deployment accuracy.

Results

Among all the models that we tested with different hyperparameters, the one with the highest validation accuracy had 96.55% validation accuracy. It was obtained by using random flipping, rotation, zoom, and translation augmentations on our custom dataset. This model was trained for 200 epochs using a StepDecay learning rate scheduler with an initial learning rate of 1e-3, the learning rate dropped every 20 epochs by a reduction factor of 0.75. After quantization, this model demonstrated a final validation accuracy of 93.10%, which means one more wrongly classified image as compared to the pre-quantization model. The deployed model had a final parameter count of 4,44,344. The deployed model also performed well in real-world testing.

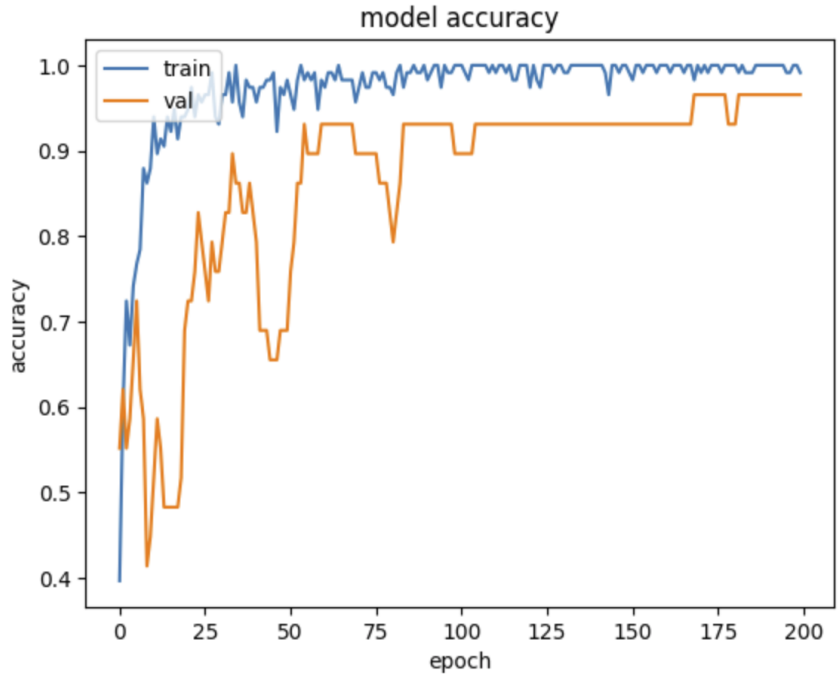

Figure 11: Graph of Training and Validation Accuracy vs Number of Epochs

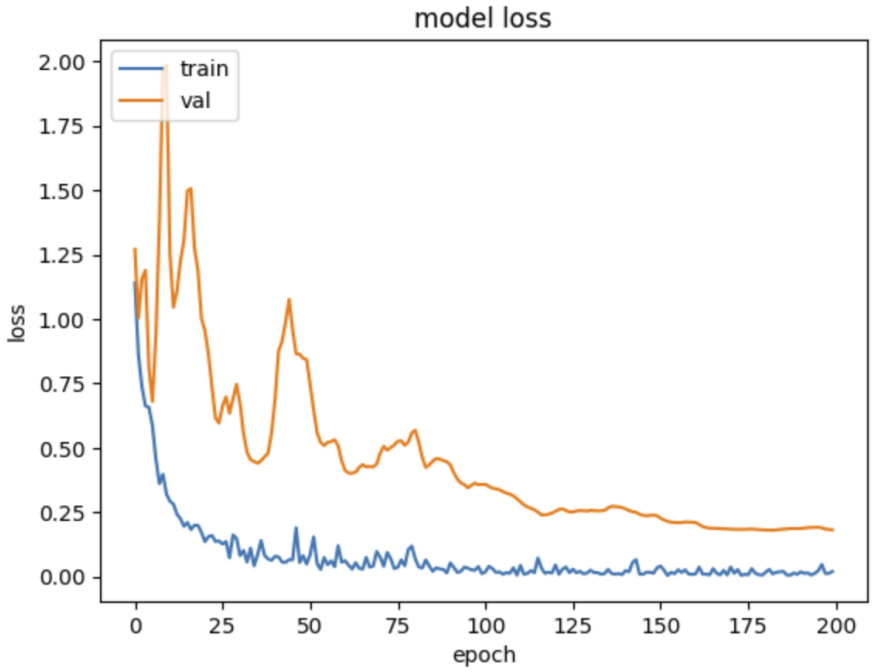

Figure 12: Graph of Training and Validation Loss vs Number of Epochs

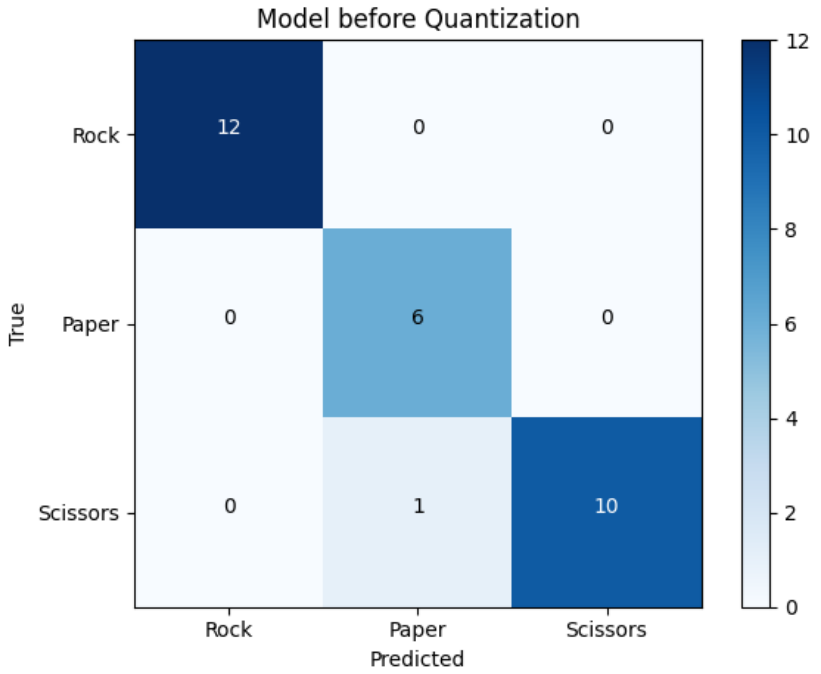

Figure 13: Confusion Matrix for the Trained Model

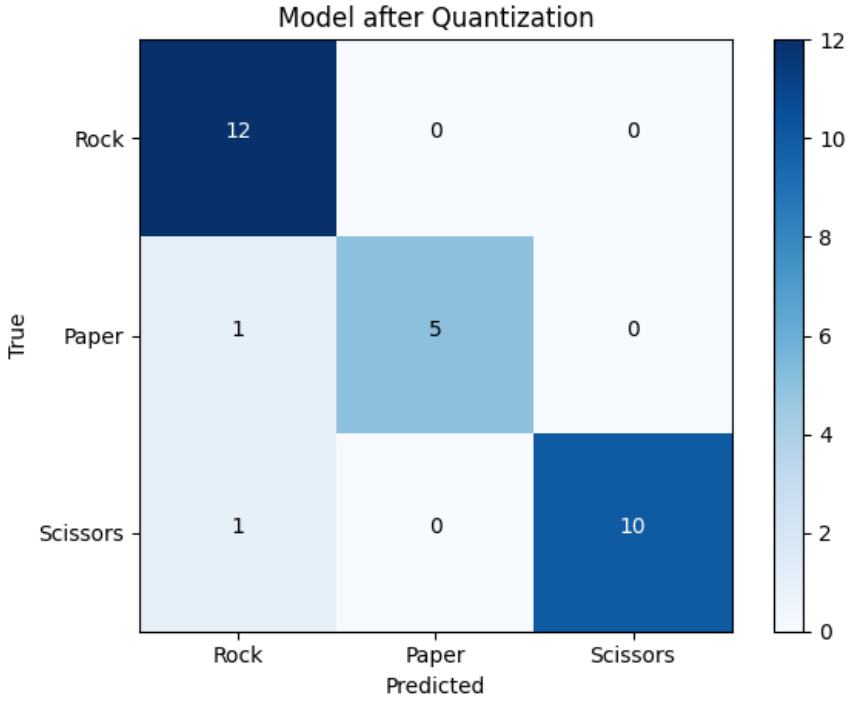

Figure 14: Confusion Matrix for the Quantized Model

Applications

One prominent application of computer vision on microcontrollers is object detection. Microcontrollers equipped with computer vision capabilities can detect and track objects of interest, enabling the development of applications such as surveillance systems, security cameras, and smart home devices. These systems can efficiently analyze visual data to identify and monitor objects, enhancing security and enabling proactive responses.

Gesture recognition is another compelling domain where computer vision on microcontrollers finds application. By leveraging computer vision algorithms, microcontrollers can interpret and recognize human gestures, enabling intuitive and interactive interfaces. This technology is particularly relevant in the fields of wearable devices, gaming consoles, and robotics, where gesture-controlled interactions enhance user experiences and facilitate natural human-machine interactions.

The integration of computer vision with microcontrollers also contributes to advancements in robotics. Microcontrollers equipped with computer vision capabilities enable robots to perceive and analyze their surroundings. They can detect objects, navigate environments, and perform tasks with increased precision. This empowers robots to autonomously carry out tasks in industrial automation, autonomous vehicles, and drones, enhancing their capabilities and enabling them to interact more intelligently with their environment.

Environmental monitoring is another area where computer vision on microcontrollers plays a vital role. By leveraging image analysis techniques, microcontrollers can process visual data to monitor and analyze changes in the environment. This capability has applications in wildlife monitoring, air and water quality assessment, and environmental conservation efforts. Microcontrollers enable the development of compact and low-power monitoring systems that can operate in remote or resource-limited areas.

In the medical field, computer vision on microcontrollers contributes to medical imaging applications. It facilitates the analysis of medical images, such as X-rays or ultrasound scans, on resource-constrained devices. This enables rapid diagnosis, telemedicine in underserved regions, and portable medical imaging solutions.

Quality control and inspection processes in manufacturing industries also benefit from computer vision on microcontrollers. Microcontrollers can employ computer vision algorithms to detect defects, measure dimensions, and ensure product compliance. This enhances efficiency, reduces human error, and improves the overall quality of manufacturing processes.

Furthermore, the application of computer vision on microcontrollers contributes to the development of intelligent Internet of Things (IoT) devices. These devices leverage visual data to perceive and interpret their surroundings. They can autonomously respond to visual cues and optimize user experiences in applications such as smart cameras, smart agriculture, smart retail, and home automation.

Conclusion

Through this project we successfully deployed a real-time and accurate gesture recognition CNN model on Arduino Nano 33 BLE Sense Lite using TensorFlow Lite for Microcontrollers. By fine-tuning the MobileNet-v2 model on a custom dataset of Rock, Paper, Scissors gestures, the potential of deep learning integration on edge devices is demonstrated, opening up opportunities for interactive and portable applications in various domains. The dataset was prepared with a Python script, ensuring suitable images for training and utilizing essential packages for image processing and communication. Image augmentation, optimized model architecture, and quantization techniques improved efficiency without compromising accuracy. Challenges were addressed iteratively, leading to enhanced dataset quality, code refinement, and improved memory usage.

Future prospects for this project involve expanding the capabilities of the Arduino Nano 33 BLE Sense Lite to include a wide range of exciting applications. From object detection and tracking in surveillance systems to gesture recognition in wearable devices and robotics. Leveraging the capabilities of platforms like the Arduino Nano 33 BLE Sense Lite can lead to groundbreaking advancements in environmental monitoring, medical imaging, quality control in manufacturing, and the development of intelligent IoT devices. Using the knowledge and experience gained from our gesture recognition model, a new CNN recognition model can be developed to classify the novel task of fabric pattern detection for the blind, which can greatly improve the lives of visually impaired individuals and promote greater accessibility in their daily activities. The possibilities for the future of computer vision on microcontrollers are limitless, promising further innovation and transformative applications across various industries.

References

1. Vijay Janapa Reddi & Pete Warden, Fundamentals of TinyML, HarvardX, edX: https://learning.edx.org/course/course-v1:HarvardX+TinyML1+1T2023/home
2. Vijay Janapa Reddi & Pete Warden, Applications of TinyML, HarvardX, edX: https://learning.edx.org/course/course-v1:HarvardX+TinyML2+1T2023/home
3. Vijay Janapa Reddi & Pete Warden, Deploying TinyML, HarvardX, edX: https://learning.edx.org/course/course-v1:HarvardX+TinyML3+1T2023/home
4. *What's coming in TensorFlow 2.0 — The TensorFlow Blog*. (2019, January 14). The TensorFlow Blog. Retrieved June 21, 2023, from https://blog.tensorflow.org/2019/01/whats-coming-in-tensorflow-2-0.html
5. Naughton, R., & Iodice, G. M. (2022). *TinyML Cookbook: Combine Artificial Intelligence and Ultra-low-power Embedded Devices to Make the World Smarter*. Packt Publishing.
6. Mohan, P., Paul, A. J., & Chirania, A. (2020). *A Tiny CNN Architecture for Medical Face Mask Detection for Resource-Constrained Endpoints*. doi:10.1007/978-981-16-0749-3_52
7. Banbury, C., Zhou, C., Fedorov, I., Matas, R., Thakker, U., Gope, D., … Whatmough, P. (2021). MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers. In A. Smola, A. Dimakis, & I. Stoica (Eds.), *Proceedings of Machine Learning and Systems* (Vol. 3, pp. 517–532). Retrieved from https://proceedings.mlsys.org/paper_files/paper/2021/file/c4d41d9619462c534b7b61d1f772385e-Paper.pdf
8. *Arduino Tiny Machine Learning Kit — Arduino Official Store*. (n.d.). Arduino Store. Retrieved July 20, 2023, from https://store.arduino.cc/products/arduino-tiny-machine-learning-kit

Glossary

API - Application Programming Interface

CMOS - Complementary Metal-Oxide-Semiconductor

CNN - Convolutional Neural Network

CPU - Central Processing Unit

DNN - Deep Neural Network

FPU - Floating Point Unit

GPU - Graphics Processing Unit

IoT - Internet of Things

MCU - Microcontroller Unit

NAS - Neural Architecture Search

OS - Operating System

QAT - Quantization Aware Training

QQVGA - Quarter Quarter Video Graphics Array

QVGA - Quarter Video Graphics Array

RAM - Random Access Memory

ReLU - Rectified Linear Unit

ROM - Read Only Memory

SRAM - Static Random Access Memory

TF - TensorFlow

TFLM - TensorFlow Lite Micro

TPU - Tensor Processing Unit

UUID - Universal Unique Identifier

VGA - Video Graphics Array