Project 2: Rock-Paper-Scissors Gesture Classification on Arduino Nano 33 BLE Sense

https://github.com/it-teaching-abo-akademi/edge-computing-for-ml-2025-ZakariaBouzada

1. Introduction

This project explores deploying a deep learning-based image classification model on an ultra-low-power microcontroller: the **Arduino Nano 33 BLE Sense**. We aimed to recognize hand gestures representing rock, paper, and scissors using a camera module (OV7670) as input and a **TensorFlow Lite Micro** model for inference.

The motivation for this project stems from the growing relevance of **Edge Machine Learning (EdgeML)**, where computations are moved closer to the data source (the "edge") instead of relying on cloud-based processing. This has many benefits, including reduced latency, improved privacy, and increased energy efficiency—key aspects in domains such as wearables, IoT, and real-time control systems.

Given the strict resource constraints of the Arduino Nano 33 BLE Sense (256KB SRAM, 1MB Flash), this project also emphasized **model compression techniques** to minimize memory footprint while maintaining adequate accuracy.

---

2. Performed Tasks

2.1 Project Setup

To begin with, we installed the necessary dependencies to get the Arduino TensorFlow Library and programs related to image conversion and display. It was also necessary to install the *Arduino_OV767X* library for the camera. This provided useful example programs for capturing images with the camera. The camera was connected to the board according to the instructions. To find the pin layout of the Arduino we consulted: https://docs.arduino.cc/resources/pinouts/ABX00027-full-pinout.pdf. Figure 1 shows an image of the camera and Arduino setup.

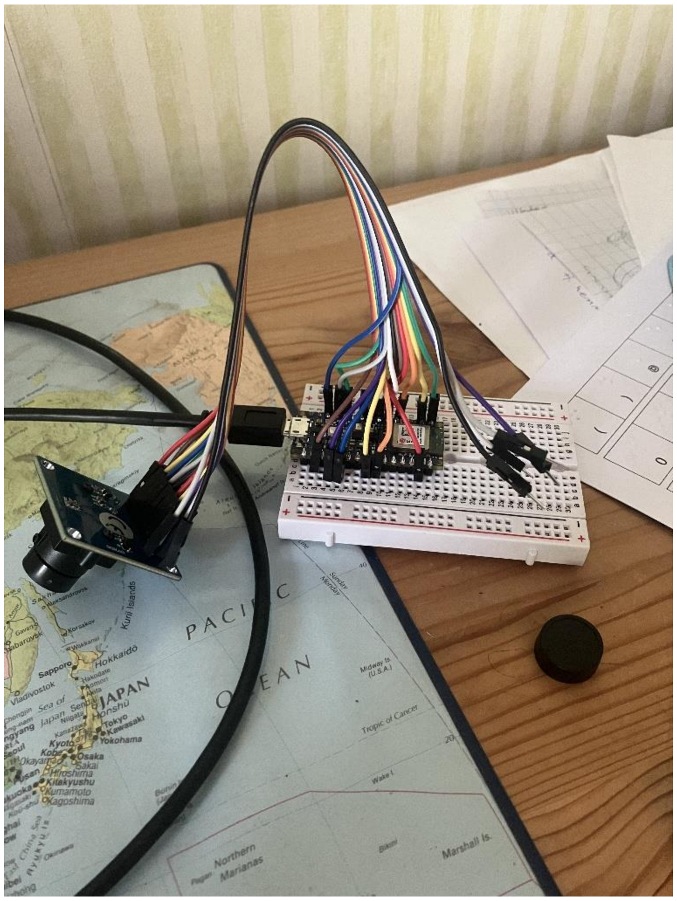

Figure 1: The Arduino and camera setup.

Following the instructions of the first lab session we captured images with the camera and displayed them using the available DisplayRawImage.py file. It was preferrable to run the Python program directly from the command prompt rather than Google Colab, since the latter doesn’t have direct access to local files. This did involve installing numpy and PIL through the pip command however. Two images captured with the camera; one colored and the other grayscale.

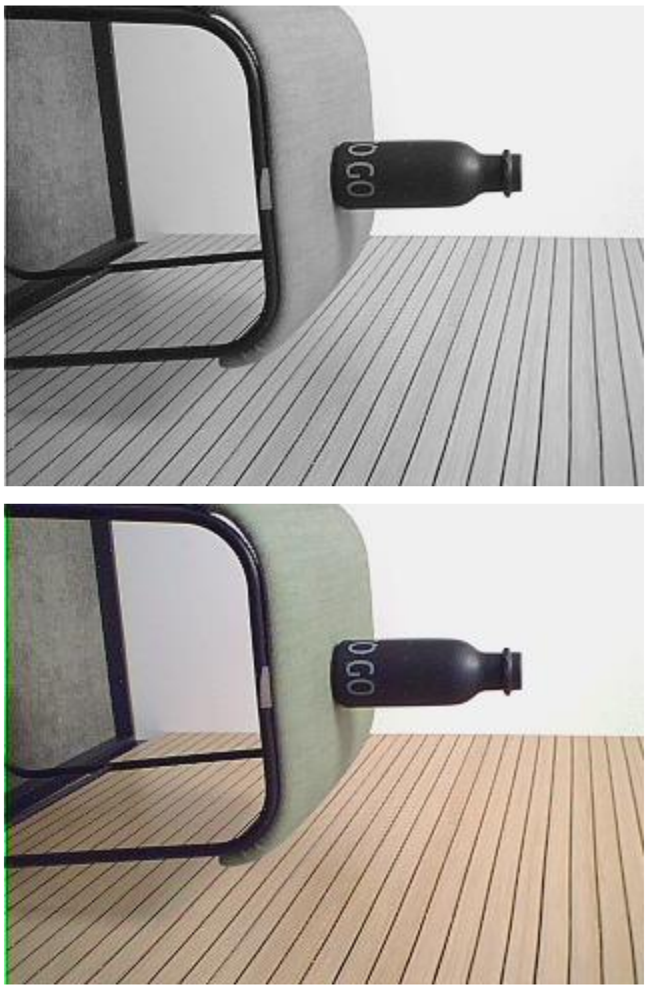

Figure 2: A grayscale and color images captured with the Arduino-attached camera.

2.2 Training The Neural Network

While working on the neural network we relied on the *EdgeML25.ipynb* file and the provided tutorials. We sourced rock-paper-scissors gesture images from the publicly available Rock-Paper-Scissors dataset and trained a convolutional neural network on this image set.

- All images were:
    - **Resized to 32×32 pixels** (compatible with the model and Arduino limitations).
    - **Converted to grayscale**, reducing the input channel from 3 (RGB) to 1.
    - Normalized to pixel values between 0–1.

We ensured balanced class distribution (~400–500 samples per class) and split the data into **training (80%)** and **validation (20%)** sets.

2.3 Model Design and Training

- Implemented a compact **CNN architecture** using TensorFlow:

```
model = keras.Sequential([
    keras.layers.Conv2D(8, (3,3), activation='relu', input_shape=(32,
32, 1)),
    keras.layers.MaxPooling2D(2, 2),

    keras.layers.Conv2D(16, (3,3), activation='relu'),
    keras.layers.MaxPooling2D(2, 2),

   keras.layers.Conv2D(32, (3,3), activation='relu'),
    keras.layers.MaxPooling2D(2, 2),

    keras.layers.Flatten(),
    keras.layers.Dense(32, activation='relu'),
    keras.layers.Dense(3, activation='softmax')
])
```

- Trained the model for 8 epochs with early stopping, using pruning-aware training.
- Achieved **<u>91% validation accuracy</u>** with the base model.

2.4 Model Compression

To fit the model on the Arduino, we applied the following compression strategies:

(a) Post-training Quantization:

- Reduced weights and activations from `float32` to `int8`.
- Converted using:

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.int8]
```

Whereas the original model had a .h-file size of around 300kB, quantization techniques were used to reduce the model size to around 100kB and even less (44kB). Dynamic range quantization was able to reduce the model size somewhat, but the integer quantization was the most effective at this.

(b) Pruning (Weight Sparsification):

- Introduced sparsity during training using `tensorflow_model_optimization`.
- Targeted **80% sparsity**:

```
pruning_schedule = tfmot.sparsity.keras.PolynomialDecay(
    initial_sparsity=0.30,
    final_sparsity=0.80,
    begin_step=0,
    end_step=end_step
)
```

These methods allowed us to reduce the `.tflite` model to somewhere between 100kB and 200kB in our experience. It is rather confusing since the summary() functions seems to say that the number of parameters increases. It would seem that the final tflite model as a .h-file is smaller in size none the less.

2.4 Model Conversion and Deployment

A test image was created from an image online, and we saved a couple of our pruned or quantized models for testing on the Arduino in deployment. In our final submission we are only including our first proper model (model3.h ~90kB) and our final model (model4.h ~44kB).

- Converted the final `.tflite` model into a `.h` header file using `xxd`:

  ```
  xxd -i model.tflite > model.h
  ```

- Integrated the model into Arduino code using the **TensorFlow Lite Micro library** and a sketch based on the IMU classifier example.
- Interfaced the OV7670 camera with the Arduino and captured frames to feed into the model.
- The live predictions (e.g., `Prediction: Rock`) were sent to the serial monitor.

2.4.1 Details of The Deployment Process

At first the IMU_classifier example was rather intimidating, but eventually it seemed fairly simple how to run our model on the Arduino. The various sensors used in the IMU program was removed from the code as they had no purpose in our project. Following the instructions, the error handler was also removed. The inference is run by simply running an invoker. The results are available in an output tensor. However, during the coding process we have encountered various errors which we have needed to solve. Firstly, it was necessary to figure out the size of the tensor arena. At first this was set to about 90kB. When we tried to run our model with the described structure, the code was compiled and written to the Arduino, but after a while it seemed to get stuck, and Windows even prompted us with an error popup about the Arduino being disconnected or similar. Through the use of the leds as debugging guides, we were able to figure out that the problem occurred inside the invoked inference of the model. Our first assumption was that the model was too large, but once we obtained a model under the size of 50kB the problem persisted. It turned out that the problem was inside the memory copying in the setup() function. Instead of reading the input tensor as 8-bit data, which our model was configured to do, we were trying to read it as float (tflInputTensor->data.f). When we changed this, all our models of reasonable size were functional. The final step was to integrate the camera into the program. This was done by using the example program which captures an image. We changed the settings to grayscale and did our own resizing method for converting the image from 176x144 resolution to 32x32. Unfortunately something caused half of the grayscale image to be recorded as 0s, and this is an issue we were unable to solve in time for the submission of this report. A final mention was the Serial.println() function which printed the prediction value of the output tensor. It had an argument 6, but this was only used for floating point accuracy. Since we were using uint8 values, the argument (6) caused our initial outputs to be outside the valid 8-bit range. By removing the argument we got rid of this numerical issue.

---

3. Results

We concluded that quantization was more effective at reducing the size of the model compared to pruning. The pruning appeared to increase the size even though this was not the case in reality. Dynamic quantization was less effective than full integer quantization. We did not retrain the models after pruning or quantizing them. Due to the issue with the camera recording only capturing half the screen, we have been unable to test the model and program in proper practice. The model responds with the same prediction every time:

```
rock: 139

paper: 66

scissor: 179
```

Table 1. Some of the noticed results.

Metric	Value
Base model accuracy	~90%
Quantized + Pruned accuracy	-
Model size (original)	~300kB
Model size (compressed)	~40-100 KB
Flash usage	351092 bytes

---

4. Discussion and Conclusions

This project illustrates the practical feasibility of deploying ML models on microcontrollers with tight memory and compute constraints. Quantization and pruning were essential to meet hardware limits. Even with a model of size 100kB and tensor arena set to 130*1024 bytes, the available memory was almost running out (>95% of availabe). Without them, the model would not fit on the device. Even though we have little to show as an end result, we have learned new things along the way. It is important to pay attention to details while working with limited resources, and C-code. Different data types, image sizes and color modes need to be accurately handled to avoid getting stuck with errors during compilation or running.

5. Reflection

This project served as an insightful and challenging introduction to **EdgeML deployment workflows**. We gained hands-on experience in:

- Model design with hardware constraints in mind.
- Applying **ML optimization techniques** for real-world embedded deployment.
- Debugging across software (Python/TensorFlow) and embedded systems (Arduino/C++) layers.
- Understanding trade-offs between **accuracy, latency, and memory** in constrained environments.

The end-to-end process, from dataset collection and model training to compression and deployment, showcased the potential and limitations of AI on microcontrollers. This project deepened our appreciation for embedded ML and the careful engineering required to make it work reliably in real-time.

1. Have you learned anything new?
   This project involved many different steps, from the installation of software through the command line, connecting the camera to the Arduino board, working with keras models in Python and writing an application in C for the Nano board. One of the big takeaways is the hands-on experience this gave us with pruning and quantization. Seeing the workflow from a standard keras model to a .h-file for tflite made us understand that there are good tools available for this kind of work.
2. Did anything surprise you?
   The tools were good. The camera was surprisingly good in terms of resolution in certain images. Sometimes it seemed to produce a Windows 2000 kind of rendering of the image, and sometimes it was as good as a new phone (it seemed). Why the camera decided to leave half of the image as 0s remained a mystery.
3. Did you find anything challenging?

Understanding much of the code was a challenge as these things were fairly new to us. For example, how the pruning seemed to increase the number of parameters. Also, regardless of the way we changed the sparsity parameters it seemed to say the same number of parameters. Perhaps the real impact is only visible during the conversion to model.h file. Making a home made image resizing function was interesting.

4. Did you find anything satisfying?
   This project was complete. It was sufficiently challenging, and gave insights into various things. Seeing oneself in the image taken with the camera was rather fun.

Appendix: Files Submitted

- Final quantized + pruned `.tflite` model
- C header file for deployment
- Arduino sketch + source code
- Full GitHub repo: https://github.com/it-teaching-abo-akademi/edge-computing-for-ml-2025-ZakariaBouzada

Zakaria Bouzada – 2102782

Hannes Kullman - 2003002