POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Technical Report

Demonstration of a Simple Transformer Running on the NPU of an STM32N6

088949- ADVANCED COMPUTER ARCHITECTURES (SILVANO CRISTINA)

Author: **Tito Nicola Drugman 10847631, Patrizio Acquadro 11087897**

Advisor: **Prof. Silvano Cristina**

Co-advisor: **Dott. Marco Ronzani**

Academic year: **2024-2025**

1. Introduction

This report represent our *Advanced Computer Architectures* project, which demonstrates the **end-to-end deployment** of both convolutional and Transformer networks on the resource-constrained *STM32N6570-DK*. After validating the tool-chain with a *Model Zoo* MobileNetV2 and a hand-crafted CNN, we design, train, INT8-quantise, and run a 4.6M-parameter encoder-only Transformer on the board, collecting detailed latency, memory, and accuracy metrics to compare the compute patterns of *CNNs vs. Transformer.*

2. Objectives and Baselines

2.1. Project Objectives

This project consists in deploying deep neural networks on the STM32N6570-DK board, starting with CNNs to validate the toolchain, to then move to the core part of the project: the Transformer deployment. In particular:

Objective 1 – Deploy Model Zoo CNN. Deploy and profile the STM32 Model Zoo network `mobilenetv2_0.35_128_quant` on the board for image classification. This serves as a verified baseline and tests the full inference pipeline using tested pre-quantized architectures.

Objective 2 – Build and Deploy Custom CNN. Design, train and quantize a lightweight custom CNN on the same dataset used for objective 1, and then deploy it on the board. This demonstrates familiarity with STM tools for measuring performance under custom choices.

Objective 3 (Core) – Full-Stack Transformer Deployment. Implement a Transformer from scratch, train it for a token-based task, quantize it to INT8, generate C and binary files, and run inference on the board. This is the core objective, requiring complete integration from model to on-device execution.

Objective 4 – Metric Collection & Architecture Comparison. For all models, we gathered key deployment metrics (latency, throughput, memory usage, and others) and compared both quantitative results and architectural characteristics, from a computational perspective.

2.2. Hardware & Toolchain Snapshot

For hardware, we used the STM32N6570-DK board and all models were trained locally on a workstation equipped with two NVIDIA GeForce RTX 5060 Ti GPUs. For optimization and deployment analysis, we leveraged the *ST*

Edge AI Developer Cloud, a cloud-based platform that allows models to be uploaded and automatically validated on real STM32 boards hosted remotely. As local toolchain, we used *STM32CubeIDE* to generate C code, validate functionality on device and target, and produce the raw files, later converted in bin. Flashing was carried out using *STM32CubeProgrammer*. All training, quantization, deployment, and inference steps were managed via *Anaconda Prompt* (see Appedix B) using Python scripts.

2.3. Task & Dataset

For the CNN we decided to focus **supervised image classification**, because it is the most broadly supported task in the Zoo (rich set of CNN variants, published latency/memory references, ready preprocessing flows), accelerating experimentation of our deployment toolchain. In particular we leverage the public **Flowers dataset**[1] (5–classes: daisy, dandelion, rose, sunflower, tulip; ∼3.7K RGB images with varied scale, illumination and background). Images are loaded from a single folder, with an *80/20* split for training and validation (`seed: 127` for reproducibility). For INT8 quantization, we reuse the same dataset with a separate *random 20% split* to calibrate activations without being affected by augmented samples. **Preprocessing** includes *RGB conversion* to ensure 3-channel input, *aspect-ratio–preserving resizing* using nearest-neighbor interpolation to avoid distortion, and *normalization of pixel values* to the $[-1, 1]$ range using the standard $(x/127.5) - 1$ formula. This matches the expected input format of our modelzoo CNN, ensuring stable behavior. **Augmentation** applies lightweight transforms—flip, contrast (0.4), brightness (0.05), translation (25%), rotation (12.5%), zoom (25%), shear (0.0515), through *nearest interpolation* and *conservative fill modes* to simulate real-world noise, exposing the model to different variation while keeping the original label semantic. This is applied only on the training split before batching to provides sufficient diversity and to try to reduce overfitting.

[1]`https://www.kaggle.com/datasets/imsparsh/flowers-dataset`

2.4. Model Zoo CNN (MobileNetV2)

Within the Model Zoo we selected `mobilenetv2_0.35`[3] as baseline, due to its efficiency and low resource footprint, since alternatives raise parameter/MAC budgets with no added methodological value.

The **architecture** is divided in 3 parts. A 3×3 stride-2 *convolutional layer*, to reduce dimensionality and generate 16 channels. *17 inverted residual blocks* with expansion ratio $t = 6$ (except the first with $t = 1$), organized in blocks $\{1, 2, 3, 4, 3, 3, 1\}$ with stride pattern $\{1, 2, 2, 2, 1, 2, 1\}$ for the first block of each group (the others have 1), and with an increasing number of channels, even if scaled by $\alpha = 0.35$ and rounded to a multiple of 8, optimizing STM32 NPU efficiency. Each block performs pointwise expansion (t×C) $\rightarrow$ depthwise 3×3 $\rightarrow$ linear pointwise projection (C), but without ReLU6 to reduce compute while preserving information. Lastly, the final part consist in a 1×1 *convolution* to get 1280 channels for a rich vector representation, followed by a *BatchNorm with ReLU6* for stable and limited activation (for quantization), followed by the *Gloabl Average Pooling* (GAP) to obtain a vector from the tensor, that is mapped to 5 logits by a *dense layer with softmax*, to produce the prediction for each clas.

Training was conducted from ImageNet-pretrained weights using Adam optimizer (lr=0.001), batch size 64, and 100 epochs, with dropout (0.3), ReduceLROnPlateau and EarlyStopping to enhance convergence.

Quantization was performed post-training using TensorFlow Lite with per-channel INT8 weights, symmetric activation ranges, and no optimization passes, producing a compact and fully STM32-compatible binary.

2.5. Custom CNN

We designed and deployed a *custom lightweight CNN* to test our ability to run fully original models on the STM32 board, instead of relying only on Model Zoo architectures already known to work. This was a crucial step to gain hands-on experience with the complete workflow, in preparation for Objective O3, which requires doing the same with a Transformer architecture.

As for the MobileNetV2, the architecture is divided into different parts. A 3×3 stride-2 convolutional layer with BatchNorm and Relu,

but which leads to 32 channels since we've $\alpha = 1$. Differently from the modelZoo CNN, we've one depthwise-separable block outputting 64 channels, followed by a maxPooling for further spatial reduction. This is followed by an inverted-residual mini-stack: two blocks to return 96 channels (expansion $t = 2$, stride 1), and two blocks for 128 channels (first stride 2, second stride 1; both with $t = 3$). Each of these blocsk perform the same three steps of the ones in MobileNetV2. However, lower expansion factors ($t = 2, 3$ vs 6) and a reduced block count shrink intermediate activation tensors (peak SRAM) and MACs while preserving essential depthwise/pointwise patterns for NPU offload. The final part is the GAP, followed by the 5-class dense softmax layer for classification.
For **training** the model reuses exactly the same task, dataset preprocessing and augmentation pipeline, optimizer and callback strategy as the Model Zoo baseline, differing only in dropout (0.25 here vs 0.30 there) and the narrower internal expansions. We also reused the same calibration subset and parameters for the post-training INT8 per-channel (weights) symmetric **PTQ**, for a direct comparison with the other baseline. Modelzoo CNN and our custom CNN can be seen being compared in Subsection 4.5.

3. Transformer

One of the core aspects of the project is the custom **encoder-only Transformer** for token-level generation. Starting from a single-token prompt, the model predicts a new token autoregressively, based on the preceding context, and repeats this process to generate entire sentences.

3.1. Architecture

We implement an encoder-only Transformer with **∼4.56M parameters**, operating on fixed-length sequences (30 tokens) with a 20K token vocabulary. Input tokens are mapped through a learnable 128-dimensional **embedding layer** over a vocabulary of 20,000 tokens. Embeddings are scaled by $\sqrt{128}$ and combined with additive **sinusoidal positional encodings**.
The core of the architecture consists of a stack of **6 Transformer encoder blocks**; each applies a *multi-head self-attention* mechanism with 4 heads, each with dimensionality 32, operating over the input representation using *learned projection matrices* W_Q, W_K, W_V, and W_O of shape 128×128. This is followed by a position-wise **feed-forward network** composed of *two linear layers* with an intermediate dimensionality of 1024 and *ReLU* activation (linear 128→1024, ReLU, linear 1024→128). Each of the MHA and FFNN sub-layers within the block are wrapped with *residual connections* and subsequent *layer normalization*, as in the original Transformer paper. However, no dropout or causal masking is embedded. The **final linear projection** (dense 128→20K) outputs logits that is dequantized and passed through a softmax to obtain a 20'000-entry probability array over every possible next token.
Parameter composition is dominated by the *attention*, since for each of its 6 layers we've 4×128^2 projection matrices, and by the *FFN* with $128 \times 1024 + 1024 \times 128$ terms, although also the *embedding and output matrices* (each 20K×128). These values were chosen to meet our perplexity and performance targets while remaining within the flash and SRAM limitations of the STM32 microcontroller.

3.2. Tokenizer

We evaluated two tokenization techniques.
Character tokenization technique splits each word at character-level. It builds a vocabulary of **66 tokens**, each mapped to a unique integer ID. These tokens are *punctuation and symbol* characters, *digits*, and the complete case-sensitive Latin alphabet (*A–Z plus a–z*), without any dedicated PAD or OOV tokens. Therefore the resulting embedding matrix is very small (66×128), leading to a negligible impact on the flash and SRAM with respect to the model, and also we can generate any word with these symbols. However, due to this limited coverage, the model requires multiple runs to generate even a single word, leading to an higher latency and higher memory. Moreover, since the model size is limited to fit the board constrains, it rarely generates meaningful words and sentences, since it does not learn effectively the semantic and syntax.
Thus, we decided to follow a **word tokenization approach**. Due to time and space reasons we decided to limit the vocabulary size to the $20'000$ *most frequent entries* (in decreasing order

of frequency), a number that we considered reasonable for a basic English comprehension [1, 2]. The out-of-vocabulary words were replaced with ID=1. As a result, the embedding matrix size increases to $20'000 \cdot 128$, which is more than 300 times the character tokenizer. The main advantage is that each generation step emits a full lexical unit instead of one character, improving effective context utilization and reducing attention cost. However, this method introduces sparsity and inefficiencies due to redundant or rare words, and suffers from limited generalization, as OOV words are collapsed into a single token. Additionally, the embedding matrix impacts the parameter count, limiting slightly the model's depth and expressiveness.

3.3. Training and Evaluation Dataset

For **training**, we used texts from Project Gutenberg[2], which hosts over 75'000 eBooks. Specifically, we selected four classic literary works:

- *The Complete Works of William Shakespeare*[3]
- *Moby Dick* by Herman Melville[4]
- *Pride and Prejudice* by Jane Austen[5]
- *Frankenstein* by Mary Shelley[6]

We merged them into a single '.txt' file using a custom Python script, that is 7.68 MB in size and contains around 1'400'000 words.

For the model's **evaluation** instead, we used an unseen corpus: *The Adventures of Sherlock Holmes* by Arthur Conan Doyle[7].

3.4. Full Stack Deployment Pipeline

This section outlines the end-to-end process for deploying our Transformer on the board, meeting its memory and performance constraints, and thus demonstrating the project's feasibility.

Training is executed on a workstation with 2 NVIDIA RTX 5060 Ti, 32GB VRAM total, Tensorflow 2.15 and CUDA 12.9. The pipeline ensures full determinism and reproducibility across dataset shuffling and weight initialization, by fixing the seed. `train()` is triggered within workflow chains (`chain_tqe`) after preprocessing and dataset split (90-10), with parameters loaded from `user_config.yaml` using `get_config()`. In particular we used the Adam optimizer with learning rate 10^{-3} and batch size 128, producing input tensors of shape $128 \times 30 \times 128$. A maximum of 100 epochs is allowed, with early stopping monitoring `val_loss` (patience 5, restore best weights), ensuring the final model corresponds to the point of lowest validation loss rather than the final training step. No mixed-precision training is applied to maintain accurate activation distributions for INT8 post-training quantization.

When we move to a resource-constrained microcontroller, `float32` are often impractical, so we decided to use **INT8 quantization**, initiated by the automated chain logic, once training completes. The procedure uses a reserved dataset subset, drawn from the same distribution as the training data but separated to avoid any augmentation, in order to collect the activation ranges for accurate quantization during inference. The function `quantize(cfg, quantization_ds, float_model_path)` performs full-model quantization with per-channel, symmetric INT8 encoding for both weights and activations, producing a deployable TFLite model and quantization metrics and deltas.

Evaluation It is important to note that this evaluation is performed entirely on device, not on the target board, serving as a critical validation step before C code generation and board deployment.

The `.tflite` model from quantization is then imported into **STM32CubeIDE** by creating a new project targeting the `STM32N6570-DK` board. The `X-Cube-AI` software pack is configure, enabling cores and setting the application to `Validation`. Once clock-related warnings are resolved, the model is loaded under the "Network" section for analysis and validation. Saving the project automatically generates both the **main.c** inference source file and the `.raw` model weights file, which is renamed to **raw.bin**, to obtain the binary file. For the **embedding matrix**, a dedicated C script (`toraw.c`) is used to sequentially extract all 20,000 quantized vectors of 128 values from `embedding_matrix.npy`, producing the corresponding `.raw.bin` file. Alter-

[2]`https://www.gutenberg.org/`

[3]`https://www.gutenberg.org/files/100/100-0.txt`

[4]`https://www.gutenberg.org/files/2701/2701-0.txt`

[5]`https://www.gutenberg.org/files/1342/1342-0.txt`

[6]`https://www.gutenberg.org/files/84/84-0.txt`

[7]`https://www.gutenberg.org/ebooks/1661`

natively, the c code can be obtained via *Cloud* or by adapting *Hello_ World* file from the documentation. This file instantiates the inference system via the macro, defining the quantized model instance, with all the buffers and structures, ensuring deterministic memory layout and latency. `init_external_memories()` configures the XSPI interface to map pre-flashed `weights.bin` and `embeddings.bin` into memory, enabling runtime access while preserving internal SRAM for activations and stack. `NPU_Config()` and `init_dwt()` configure the NPU and activate the DWT-based cycle counter for precise performance profiling. As the model lacks a final softmax layer, a custom implementation (`softmax_with_temperature()`) handles this on-device: INT8 logits are dequantized, scaled by temperature (1), normalized using $\exp(z - \max z)$, and the next token is sampled.
Deployment involves flashing two binaries onto the board's external XSPI flash: the quantized Transformer weights at address `0x71000000` and the embedding table at `0x71470000`. This memory-mapped layout allows direct access without runtime relocation and enables modular updates (e.g., changing only embeddings). To do so we need to launch *STM32CubeProgrammer* and connect the *board via USB-C*. In the "Erasing Programming" tab, we select and flashes both binaries at the specified addresses, by clicking on *Start Programming*. After this we disconnect the board and close the app, and start a debug build in STM32CubeIDE. After the success message, pressing `F8` starts execution on the board. Lastly, on the host PC terminal, we run `python serial_demo.py` (after selecting the correct COM port) to run the transformer and display the model's output, confirming correct deployment and runtime behavior.
Lastly, we performed the **inference** on the board, using *fixed compile-time constants* `CTX_LEN = 30`, `EMB_LEN = 128`, and `VOCAB_LEN = 20000`. This leads to two static buffers: `g_input_tensor[30][128]` for the quantized embeddings, and `g_logits[20000]` for the final float output. Inference begins when the user selects a seed token on the host PC, whose ID is retrieved via the `word2idx` dictionary in the `tokenizer.json`. This ID is packed into 4 bytes in little-endian order and transmitted over UART to the board. On the firmware side, the bytes are reconstructed into a `uint32_t new_token` in SRAM, and the routine `init_text_generation()` clears the full context buffer with a `memset`, ensuring a clean state. The token's quantized embedding is then inserted into the final position of the buffer, while all previous positions remain zero-padded, as a left-padded context. The full context buffer is copied into `npuRAM5`, and passed to the LL-ATON runtime, which triggers a DMA transfer of the $30 \times 128 = 3840$ bytes into the first layer of the NPU. The NPU processes this input and outputs a $1 \times 30 \times 20000$ tensor of quantized logits. However, only the last timestep (index 29) is relevant, as it corresponds to the most recent context position. Its logits are extracted, dequantized (using known scale and zero-point), and passed to `softmax_with_temperature()`, which returns the next token ID. The sampled token ID is then re-packed into 4 little-endian bytes and transmitted back to the host. The host decodes the ID via `id2w`, prints the corresponding word, and re-feeds it as input for the next step by invoking `generate_one_step()`. Each step performs a full re-initialization of the runtime, followed by de-initialization, in order to profile worst-case latency. This loop repeats until 128 tokens have been generated. Notably, the system does not apply any causal attention mask, but autoregression is obtained by shifting the input window and overwriting the tail with the latest token's embedding. Moreover, sampling is performed with an unseeded PRNG, so each run produces a different output sequence, despite identical length.

4. Results

We decided to evaluate the transformer in terms of three main categories:

- **Model**: *Accuracy* and *perplexity* to assess model's performance.
- **Embedding**: *Cosine similarity* of synonyms to evaluate the embedding space.
- **Board**: To evaluate *on-board performance* and *resource usage* we tracked:
  - Weights and embeddings in the flash
  - Peak RAM usage (activations)
  - Total MACC/FLOPs
  - Hardware vs. software execution time
  - Inference latency

◦ Throughput
◦ Size of `tokenizer.json` on host (PC)

4.1. Model

We computed accuracy and perplexity over 20'000 tokens based on an unseen corpus (see subsection 3.3) and we obtained an **accuracy of 0.148** and a **perplexity of 653**. Although an accuracy of 14.8% may seem low, it is still almost $3'000\times$ better than randomly guessing one token out of a 20'000-word vocabulary. Likewise, a perplexity of 653 implies the model is, on average, as "undecided" as if choosing uniformly among 653 possible next-word candidates, which is $\frac{1}{30}$ of the full dictionary. There is clearly *ample room for improvement*, but our primary goal was simply to demonstrate that a full transformer regardless of its raw accuracy or perplexity can be deployed and run on the STM32 board.

4.2. Embeddings

To quantitatively assess the quality of the learned word embeddings, we developed a *Python script* to compute **cosine similarity** between individual tokens or pairs. This relies on the idea that semantically related words should have nearby vectors in embedding space. As shown in **Table 4**, high similarity scores confirm that the model captures basic semantic relationships essential for coherent generation, as attention mechanisms depend on them for meaningful predictions. Despite the model's limited size, it successfully learns some associations: e.g., after *You*, the closest tokens are verbs like "pause" (0.7655) and "eat" (0.7647); *Romeo* and *Juliet* score 0.6538. However, not all semantic relations are preserved. In fact, more accurate embeddings would require higher dimensionality and a larger corpus.

4.3. Board

The deployment of our quantized Transformer model on the STM32N6570-DK demonstrates efficient use of flash and SRAM. The model includes 4.56M INT8 parameters (4.59MB) and a $20k \times 128$ embedding matrix (2.56 MB), both stored in external **Octo-SPI flash** for a total footprint of 7.159MB. So just 6.4% of the 112 MB available is occupied, leaving room for expansion (transformer with more parameters and/or larger embedding). **SRAM** usage peaks at 1.208MB. The input tensor buffer `g_input_tensor[30][128]` occupies 1MB in `cpuRAM2`, while activations use 147.9kB in `npuRAM3` and 64.7kB in `npuRAM5`. This fits comfortably within the on-chip AI accelerator RAM, proving that ($d_{\text{model}} = 128$) is an effective compromise between capacity and memory pressure. Each token inference involves 172.5M **MACC** operations, with 257 epochs handled by the NPU and 150 by the CPU, for a total of 407. This hybrid execution reflects hardware acceleration for core layers, while non-NPU-compatible parts (like Softmax) fallback to software. Generating 128 tokens takes 9.361s, with an average **latency** of 73.14ms/token, yielding 13.67 tokens/s **throughput**. Notably, the runtime is reinitialized before each token (cold-start mode) to capture worst-case latency. With persistent runtime, latency could drop to 21ms/token. The tokenizer, stored only on the host PC, weighs 745KB and doesn't impact embedded resources. Overall, these results (in **Table 1** and in **Table 2**), confirm that our quantized model fits within embedded constraints while maintaining real-time generation capabilities.

4.4. Computational Comparison: CNNs VS Transformer

From a **memory** perspective, the Transformer is substantially heavier, requiring *7.16MB of flash* (4.59 MB for weights, 2.56MB for embeddings) and *1.21MB of SRAM*, while MobileNetV2_0.35 and the custom CNN fit within *600kB of flash* and *300kB of SRAM*, making them more suitable for embedded devices. In terms of **computation**, the Transformer needs over *172M MACCs* per step, versus *15.7M* for MobileNet and only *4.2M* for the CNN—up to 40×. This leads to higher energy and latency. Regarding **NPU usage**, CNNs are more hardware-friendly: MobileNet runs *57/58 epochs* on the NPU, the CNN *95%*, while the Transformer only executes *257 out of 407 epochs* on hardware, relying heavily on the CPU (e.g., for softmax). Consequently, the Transformer's **latency** is *73.1ms/token*, with a *throughput of 13.67tokens/s*, while CNNs complete inference in a few milliseconds. While Transformers provide flexibility, CNNs remain far more efficient under embedded constraints.

4.5. Results Comparison: ModelZoo CNN VS Custom CNN

We deployed and quantized two different convolutional neural networks on STM microcontroller and evaluated them on the same flower-classification dataset. Modelzoo CNN (Figs. 1–2) reached a validation accuracy of 90.87 %, exhibiting strong per-class performance (all classes above 85 %). Its training curves show smooth convergence of training loss to near zero and validation loss to about 0.5, with training accuracy climbing to 97.5 % and validation to 90 %. Our custom CNN (Figs. 3–4), in contrast, overfits: although its training accuracy saturates at 100 %, its validation accuracy plateaus around 75 % and its validation loss remains unstable with large spikes. The confusion matrix highlights especially poor performance on tulips (only 62 % correct) and roses (73 %), suggesting this model's architecture is less robust for embedded inference. Table 3 compare the two quantized models deployed on the STM32 N6 NPU. Modelzoo CNN attains markedly higher validation accuracy (90.87 % vs. 76.43 %), with a smaller generalization gap (training–validation gap $\approx$ 6.6 pp vs. 23.6 pp for Model 2), indicating better regularization and architectural suitability for the dataset. Our custom CNN, although it uses 40.5 % fewer parameters and almost half the weight memory (48.6 % reduction), allocates 140 % more activation memory, yielding a higher total memory footprint (+5.7 %). Because activation memory dominates embedded inference latency and energy (due to buffer movement), The second CNN design trades off parameter efficiency for higher intermediate feature storage—yet suffers a 14.44 pp absolute accuracy drop. The first CNN ran 58 epochs (57 entirely on the NPU plus 1 in software for the Softmax), while our custom CNN ran 20 epochs (19 on-chip, 1 for the Softmax in software). Modelzoo CNN trades off slightly larger weight memory for much smaller activation buffers, achieving a 90.87 % validation accuracy with only a 6.6 pp generalization gap. Our custom CNN halves the weight footprint at the cost of a 140 % activation memory increase and a 14.44 pp drop in accuracy, indicating overfitting and higher on-chip buffer usage.

5. Conclusion and Future Work

The deployed model reached a **accuracy** and a **perplexity** on the unseen corpus substantially better than random performance, but far from state-of-the-art. However, this performance gap aligns with expectations from the very constrained setup. Moreover, the *embedding space* showed some coherent semantic structure, hinting at the model's capacity to internalize core lexical relations even in this limited environment.
Several enhancements could be explored. Using richer corpora, increasing embedding size (128 $\rightarrow$ 256), and switching to modern, cleaner text would likely improve generation quality. *Language adaptation* to Italian would also have been a valid option, allowing us to evaluate the transformer's outputs more effectively if in our native language. Furthermore, as this project prioritized **deployment feasibility** over output quality, we intentionally limited model size and preprocessing to fit embedded constraints, but obviously increasing the number of parameters and fine tuning would improve the performance.
In summary, while this Transformer cannot yet match general-purpose LLMs, the project successfully demonstrated the feasibility of **training**, **quantizing**, and **deploying** a custom encoder-only Transformer on the **STM32N6570-DK**, despite its strict memory and performance constraints. With 6 blocks, 4 attention heads, model width of 128, and FFN hidden size of 1024, the quantized INT8 model required just **7.16MB** of Octo-SPI flash—well below the **112MB** available—while staying within the SRAM budget for activations. As discussed in *Section 4.3*, this leaves room for future scaling of embedding size or model depth, at the cost of increased inference time, paving the way for future *embedded NLP applications.*

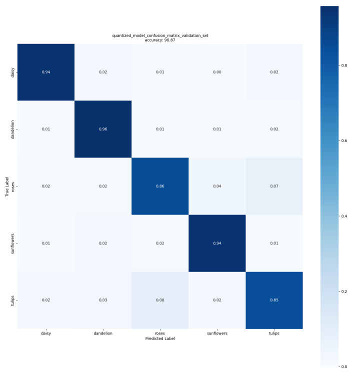

Figure 1: ModelZoo CNN confusion matrix (val. acc. 90.87 %).

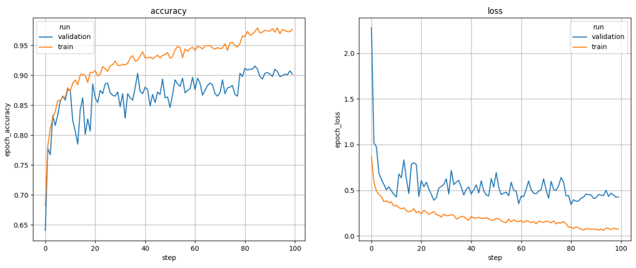

Figure 2: ModelZoo CNN training/validation curves.

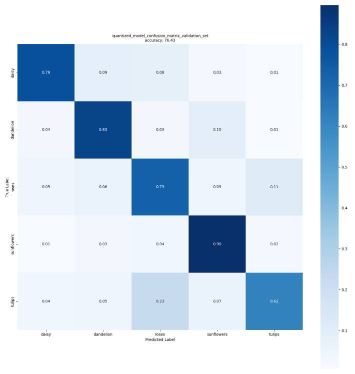

Figure 3: Custom CNN confusion matrix (val. acc. 76.43 %).

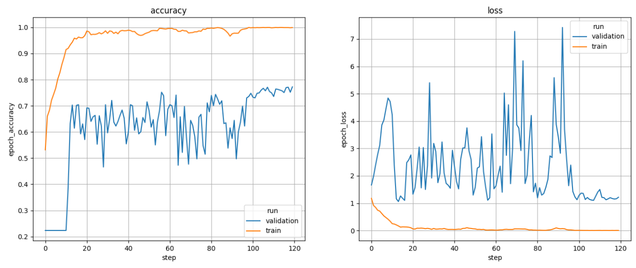

Figure 4: Custom CNN training/validation curves.

A. Cosine Similarity

In Table 4 we report some of the cosine similar we obtained.

B. Software Environment

To reproduce the exact Python environment used in this project, recreate the Conda environment from the provided `environment.yml` file, see Listing 1.

```
name: st_zoo_validate2
channels:
  - defaults
  - https://repo.anaconda.com/pkgs/main
  - https://repo.anaconda.com/pkgs/r
  - https://repo.anaconda.com/pkgs/msys2
dependencies:
  - python=3.10.18
  - pip=25.1
  - setuptools=78.1.1
  - numpy=1.25.1
  - pandas=2.0.2
  - matplotlib=3.8.0
  - scipy=1.11.1
  - scikit-learn=1.2.2
  - pytorch=2.1.0
  - torchvision=0.16.0
  - tflite-runtime=2.18.0
  - threadpoolctl=3.6.0
  - tqdm=4.67.1
  - uvicorn=0.35.0
  - waitress=3.0.2
  - werkzeug=3.1.3
  - wrapt=1.14.1
  - zipp=3.23.0
  - tzdata=2025.2
  - pip:
      - pyradiomics==3.0.1
      - SimpleITK==2.2.1
      - typing-extensions==4.14.1
      - typing-inspection==0.4.1
```

Listing 1: Conda environment specification

Metric	**Value**	**Unit / Notes**
Flash footprint (model + embeddings)	7.159	MB (quantized, external Octo-SPI)
Model weights	4.590	MB
Int8 model parameters	4’562’979	values
Embedding file size	2.560	MB
Int8 embedding entries	$20'000 \cdot 128$	values
Peak SRAM (activations + buffers)	1.208	MB
MACC	172,531,800	ops
Epochs executed	407	epochs
Epochs on NPU (hardware)	257	runs
Epochs on CPU (software)	150	runs
Tokenizer file size (host PC)	745	KB (JSON)
Generation length (reported run)	128	tokens
Time for 128-token generation	9.361	s (single representative run)
Average latency per generated token	73.14	ms/token (same run)
Throughput	13.67	tokens/s (same run)

Table 1: On-board resource usage and runtime performance (STM32N6570).

Region	Address range	Usage			Weights	Activations
		Used	Total	% Used		
flexMEM	0x34000000–0x34000000	0 B	0 B	0.00%	0 B	0 B
cpuRAM1	0x34064000–0x34064000	0 B	0 B	0.00%	0 B	0 B
cpuRAM2	0x34100000–0x34200000	1.000 MB	1.000 MB	100.00%	0 B	1.000 MB
npuRAM3	0x34200000–0x34270000	147.875 kB	448.000 kB	33.01%	0 B	147.875 kB
npuRAM4	0x34270000–0x342E0000	0 B	448.000 kB	0.00%	0 B	0 B
npuRAM5	0x342E0000–0x34350000	64.688 kB	448.000 kB	14.44%	0 B	64.688 kB
npuRAM6	0x34350000–0x343C0000	0 B	448.000 kB	0.00%	0 B	0 B
octoFlash	0x71000000–0x78000000	7.159 MB	112.000 MB	6.40%	7.159 MB	0 B
hyperRAM	0x90000000–0x92000000	0 B	32.000 MB	0.00%	0 B	0 B
Total		8.367 MB			**7.159 MB**	**1.208 MB**

Table 2: Memory usage information (including embedding file; I/O buffers are included in activations)

Metric	Model 1	Model 2	Rel. Change
Parameters (count)	395 493	235 493	−40.5 %
Weight memory (KiB)	594.3	305.8	−48.6 %
Activation memory (KiB)	240.0	576.0	140.0 %
Total memory (KiB)	834.3	881.8	5.7 %
Total number of epochs	58	20	
Training accuracy (%)	97.5	100.0	2.6 pp
Validation accuracy (%)	90.87	76.43	−15.9 %
Gen. gap (train–val, pp)	6.6	23.6	258 %

Table 3: Quantized CNN metrics on STM32 N6 NPU. Relative change is Model 2 vs. Model 1.

C. Code

The repository containing the code as well as the dataset is available on GitHub[8].

[8]https://github.com/TitoNicolaDrugman/Transformer-NPU-STM32N6

Query: You		
Rank	Token	Score
1	pause	0.7655
2	eat	0.7647
3	enough	0.7572
4	reality	0.7495
5	rumour	0.7474
6	spoke	0.7465
7	ginger	0.7443
8	brown	0.7433
9	examples	0.7370
10	commence	0.7355
Query: Romeo		
Rank	Token	Score
1	thee	0.7894
2	withdrew	0.7832
3	dice	0.7825
4	interview	0.7820
5	consists	0.7819
6	stricken	0.7808
7	roared,	0.7792
8	suspicious	0.7784
9	yield,	0.7768
10	witness	0.7746
Query: King		
Rank	Token	Score
1	Ephesus.	0.7676
2	conversed	0.7667
3	well.	0.7612
4	death's	0.7572
5	marry	0.7566
6	gravel	0.7480
7	men's	0.7480
8	sooner	0.7460
9	Master,	0.7458
10	Either	0.7450
Cosine(Romeo, Juliet)		0.6538

Table 4: Top-10 most similar tokens for each query, plus the cosine similarity between *Romeo* and *Juliet*.

References

[1] Na Liu and I. S. P. Nation. Factors affecting guessing vocabulary in context. *RELC Journal*, 16(1):33–42, 1985.

[2] Ahmad Azman Mokhtar, Rafizah Mohd Rawian, Mohamad Fadhili Yahaya, Azaharee Abdullah, Mahani Mansor, Mohd Izwan Osman, Zahrullaili Ahmad Zakaria, Aminarashid Murat, Surina Nayan, and Abdul Rashid Mohamed. Vocabulary knowledge of adult esl learners. *English Language Teaching*, 3(1):71–80, March 2010.

[3] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.