



TinyML Workshop

Building Efficient AI for the Edge.

Obed Mogaka | obed.mogaka@mdu.se

HERO Lab, Malardalens University

November 24, 2025

Overview

1. TinyML on Microcontrollers

- Challenges of TinyML.
- Applications.

2. Tiny neural network design

- MCUNet

3. TinyEngine and Parallel Processing

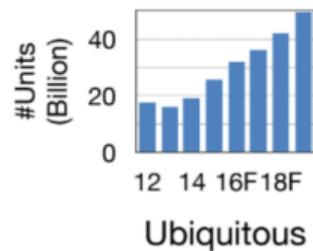
- Parallel computing techniques
- Inference optimizations

TinyML on Microcontrollers

Deep Learning Going "Tiny"

Squeezing deep learning into IoT devices

- Billions of IoT devices around the world based on microcontrollers.
- **Low-cost:** Democratize AI.
- **Low-power:** Green AI, reduce carbon



Low-cost
(\$0.1 - \$10)



Low-power
(mW)

"Tiny" Applications

- **Applications in our daily life:**

- Tiny vision
- Tiny audio
- Tiny time series/anomaly detection

- Wide range of applications.

Smart Home



Smart Manufacturing



Personalized Healthcare



Precision Agriculture



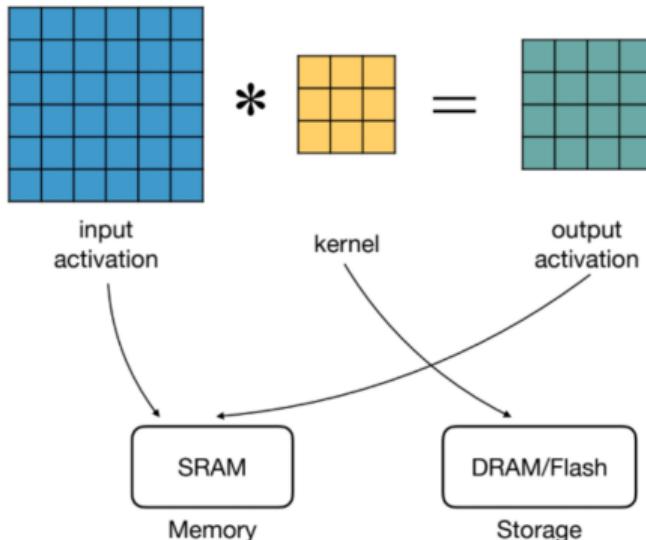
TinyML is Challenging



| | Cloud AI | Mobile AI | Tiny AI |
|---------------------|----------|-----------|---------|
| Memory (Activation) | 32GB | 4GB | 320kB |
| Storage (Weights) | ~TB/PB | 256GB | 1MB |




Running CNNs on MCUs



Arduino Nano 33 BLE Sense

- SRAM: 256KB
- Flash: 1MB



STM32 F746ZG

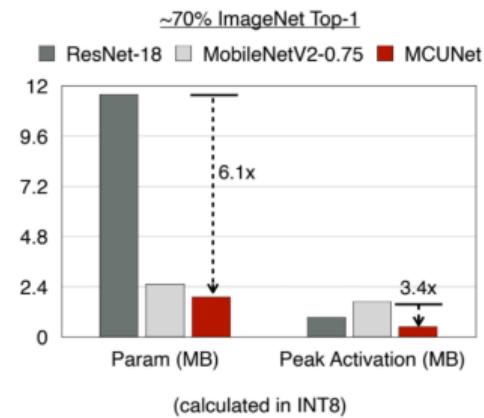
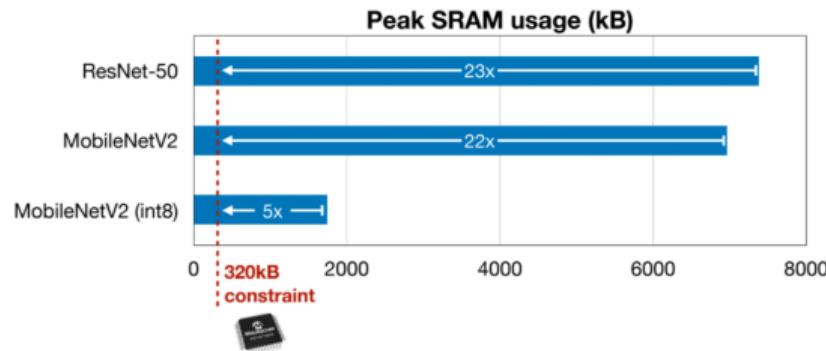
- SRAM: 320KB
- Flash: 1MB

- **Flash Usage**
 - = **model size**
 - **Static**, need to hold the entire model
- **SRAM usage**
 - = **Input activation + output activation**
 - Dynamic, different for each layer
 - We care about peak SRAM
 - (Weights are not counted since they can be partially fetched)

CNNs are too Big for TinyML!

To fit CNN on MCUs, we need lightweight architectures.

MobileNetV2 reduces only model size but not peak activation size



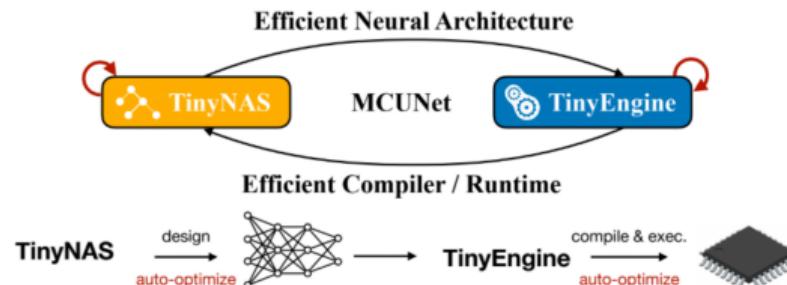
MCUNet¹ reduced not only the model size but also the activation size.

¹Lin, Ji, et al. "MCUNet: Tiny deep learning on iot devices." Advances in neural information processing systems 33 (2020): 11711-11722.

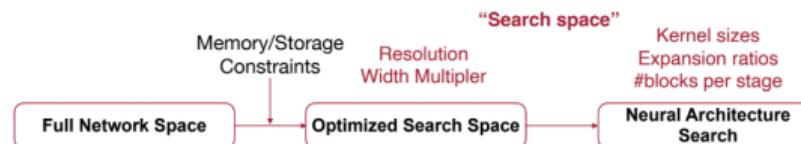
Tiny Neural Network Design

Tiny Neural Network Design

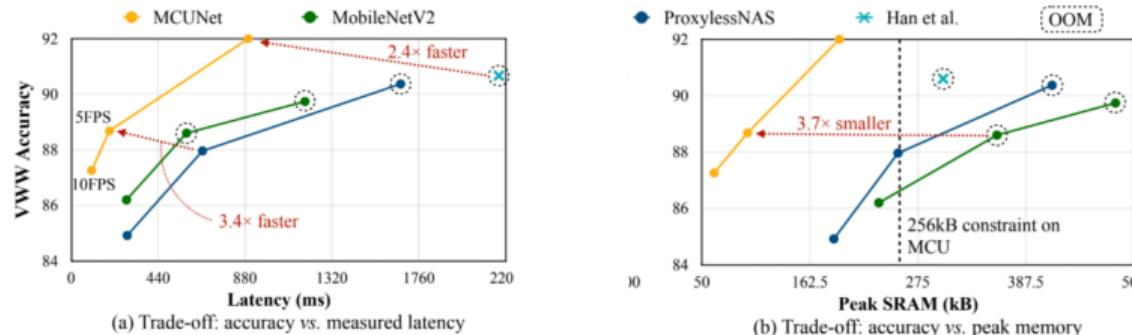
MCUNet: System-Algorithm Co-design.



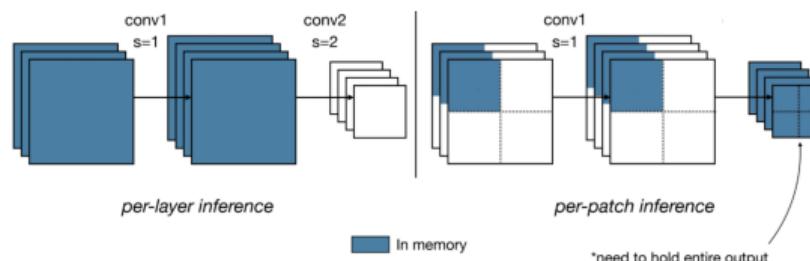
- **TinyNAS** is two-stage NAS for Tiny memory constraints.
- First design the design space, then search the sub-network



MCUNet achieves higher accuracy at lower memory. (Visual Wake Works (VWW) dataset)



MCUNetV2 introduced patch-based inference to break the memory bottleneck.



TinyEngine and Parallel Processing

Efficient Computing Techniques

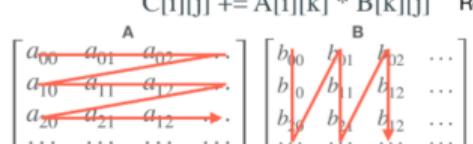
- **Loop optimization:** Optimize locality and reduce branching overhead
 - Loop reordering: Optimizes locality by reordering the sequence of loops.
 - Loop tiling: Reduces memory access by partitioning a loop's iteration space
 - Loop unrolling: Enforces parallelism
- **SIMD** (Single instruction, multiple data) programming:
 - Performs the same operation on multiple data points simultaneously.
- **Multithreading:**
 - Concurrent execution of multiple threads within a single process.

Loop Reordering

Improve data locality

- Data movement (cache miss) is very expensive
- Chuck of memory is fetched at a time (cache line)
- Reduce cache miss by loop reordering:
 - Change the order of loop iteration variables.

```
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```

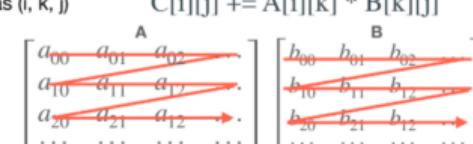

Good data locality Poor data locality!

* Assume stored in row-major order

Without Loop reordering:

naive_mat_mul: 24296 ms

```
for i in range(0, N):
    for k in range(0, N):
        for j in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```


Good data locality

* Assume stored in row-major order

With Loop reordering:

mat_mul_reordering: 1979 ms

12x speed up

*Results are measured on Intel Xeon 4114

Loop Tiling

Reduce cache miss

- What happens when data is larger than cache size?
- Data in cache will be evicted before reuse → cache miss ↑
- Loop tiling reduces cache misses:
 - Partition loop iteration space.
 - Fit accessed elements in the loop into cache size
 - Ensure data stays in the cache until it is reused

```
for i in range(0, N):
    for k in range(0, N):
        for j in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```

$$N \begin{bmatrix} b_{00} & b_{01} & \dots & \dots \\ b_{10} & b_{11} & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Accessed elements in B: N^2

→
Tile the loop of **j**

```
T_j = TILE_SIZE
→ for j_t in range(0, N, T_j):
    for i in range(0, N):
        for k in range(0, N):
            → for j in range(j_t, j_t + T_j):
                C[i][j] += A[i][k] * B[k][j]
```

$$T_j = \text{TILE_SIZE} \quad N \begin{bmatrix} b_{00} & b_{01} & \dots & \dots \\ b_{10} & b_{11} & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Accessed elements in B: $N \times \text{TILE_SIZE}$

Loop Unrolling

Reduce branching overheads

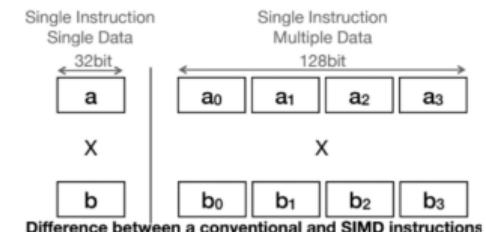
- Overheads of loop control: arithmetic operations for pointers, end of loop tests, branch predictions.
- Loop unrolling reduces these overheads and enhances parallelism:
 - Replicate the loop body a number of times.
 - Tradeoff between hardware resource usage and reduced overheads

```
for i in range(0, N):
    for i in range(0, N):
        for j in range(0, N):
            for k in range(0, N):
                C[i][j] += A[i][k] * B[k][j]
                e.g., unroll by 4
                for j in range(0, N):
                    for k in range(0, N, 4): # step 1->4
                        C[i][j] += A[i][k] * B[k][j]
                        C[i][j] += A[i][k+1] * B[k+1][j]
                        C[i][j] += A[i][k+2] * B[k+2][j]
                        C[i][j] += A[i][k+3] * B[k+3][j]
```

- Arithmetic operation for pointers: $N^3 \rightarrow 1/4N^3$
- Number of loop tests: $N^3 \rightarrow 1/4N^3$
- Code size of the innermost loop: $1 \rightarrow 4$

SIMD Programming

- SIMD (Single Instruction Multiple Data)
 - A parallel processing paradigm that applies a single instruction to multiple data elements simultaneously.
 - Commonly used in modern processors to exploit data-level parallelism
- With SIMD programming, we can:
 - Increase computational throughput and speed
 - Improve energy efficiency.



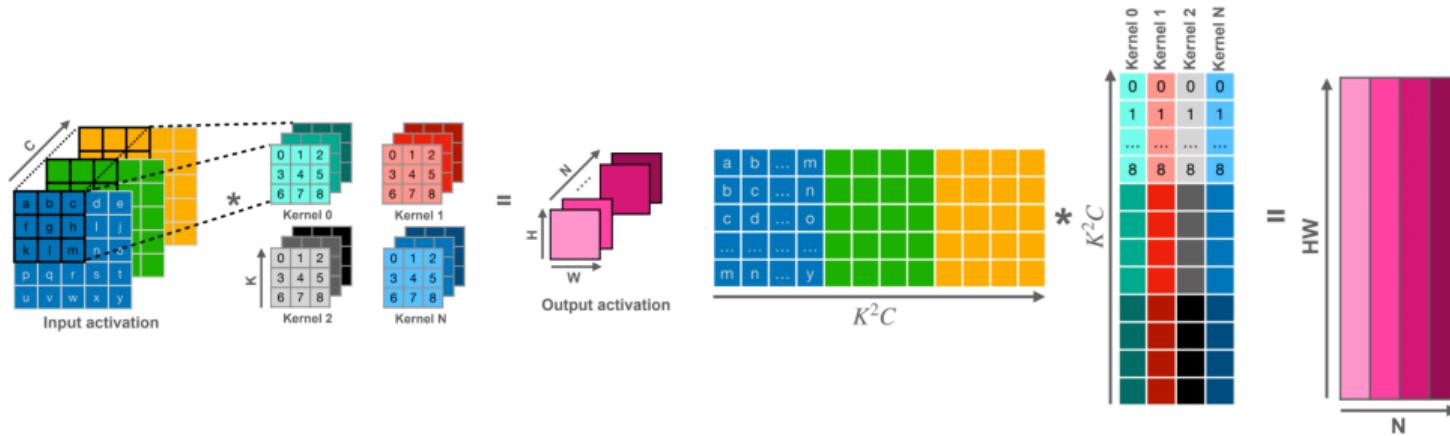
Inference Optimizations

To enhance computing speed and reduce memory usage

- Image to Column (Im2Col) convolution:
 - Rearranges input data to directly utilize matrix multiplication kernels
- In-place depth-wise convolution:
 - Reuse the input buffer to write the output data, so as to reduce peak SRAM memory.
- NHWC for point-wise convolution, and NCHW for depth-wise convolution:
 - Exploit the appropriate data layout for different types of convolution.
- Winograd convolution:
 - Reduce the number of multiplications to enhance computing speed for convolution.

Im2Col Convolution

Im2col is a technique to convert the image in a form such that utilized **Generalized Matrix Multiplication** (GEMM) kernels for dot products.

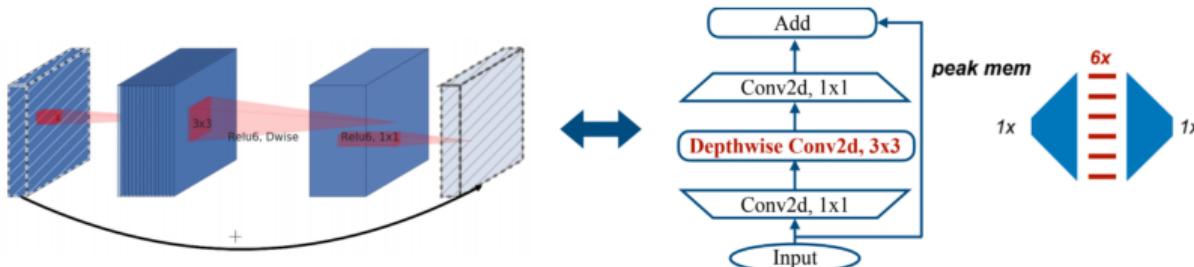


- Im2Col however requires additional memory space.

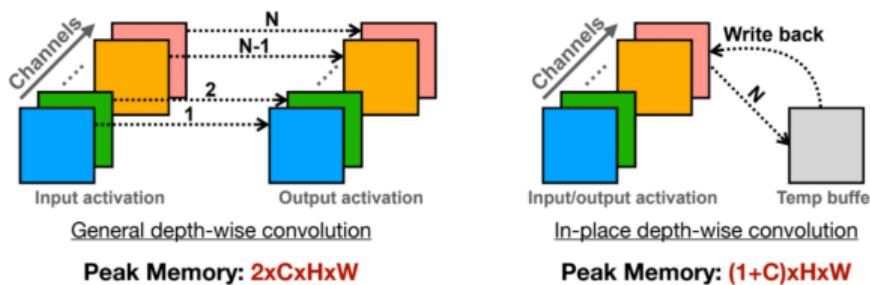
In-place depth-wise convolution

Inverted Residual Block

- Many popular DNN models, such as MobileNetV2, have "inverted residual blocks" with depth-wise convolutions which reduce model size and FLOPs, but significantly increase peak memory (3-6x)



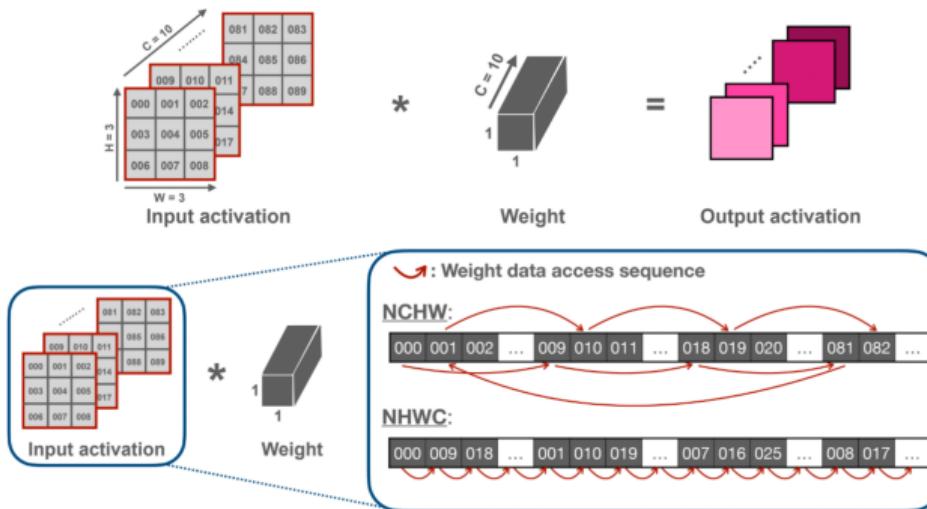
- To reduce the peak memory of depth-wise convolution, we utilize the "in-place" updating policy with a temporary buffer.



Optimal Data Layout

Use NHWC for Point-wise Convolution

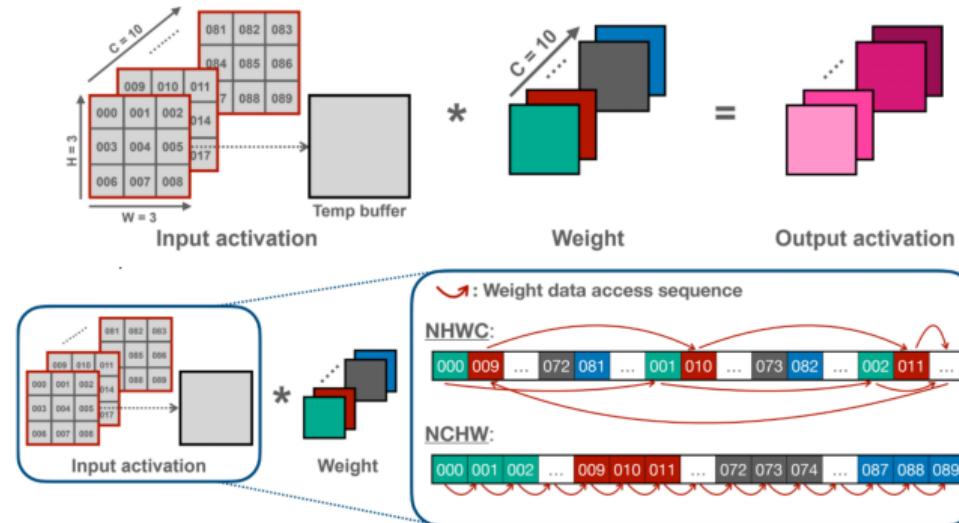
- TinyEngine adopts the NHWC data layout for point-wise convolution. Generally, NHWC would have better locality than NCHW for point-wise convolution due to more sequential access.



Optimal Data Layout

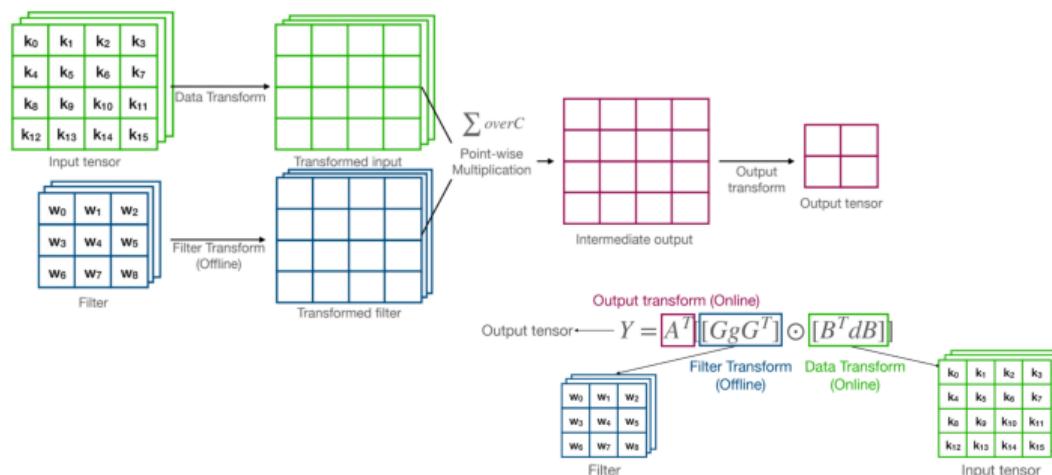
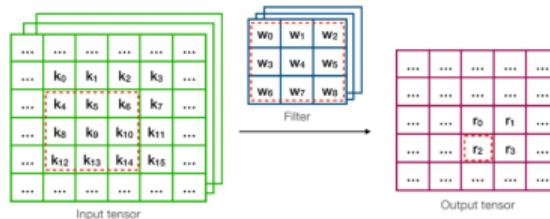
Use NCHW for Depth-wise Convolution

- TinyEngine adopts the in-place depth-wise convolution.
- NCHW would have better locality than NHWC - we will access activation and conduct depth-wise convolution in the NCHW sequence.



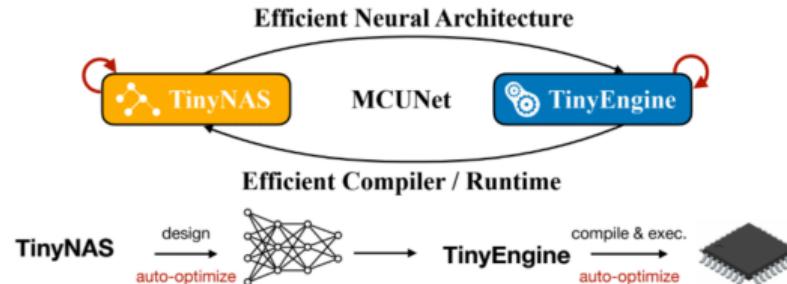
Winograd Convolution

- Direct convolution need $9 \times C \times 4$ MACs for 4 outputs
- Winograd convolution need only $16 \times C$ MACs for 4 outputs $\rightarrow 2.25 \times$ fewer MACs



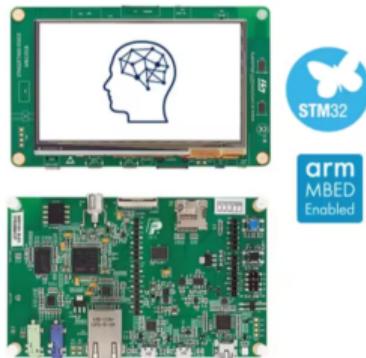
TinyEngine on MCU

MCUNet: System-Algorithm Co-design.



- **TinyEngine** is a memory-efficient and high-performance neural network library for Microcontrollers.
- TinyEngine is a part of MCUNet, which also consists of TinyNAS.
- **TinyNAS** is two-stage NAS for Tiny memory constraints.
- MCUNet is a system-algorithm co-design framework for tiny deep learning on microcontrollers.
- TinyEngine and TinyNAS are co-designed to fit the tight memory budgets.

TinyEngine on MCU



Specifications and features

- **Microcontroller**: STM32F746NGH6, based on Arm® Cortex®-M7 core.
- **Memory**: Includes 16.332 MB RAM and 17 MB Flash/SPI Flash.
- **Connectivity**: Features high-speed connectivity, enabling various applications.
- **Multimedia**: Supports audio, graphics, and video.

NEW: Arduino Uno Q



Qualcomm Dragonwing™ QRB2210

Includes the powerful Qualcomm Dragonwing™ QRB2210 processor featuring:

- Quad-core Arm® Cortex®-A53 @ 2.0 GHz
- Adreno GPU 3D graphics accelerator
- 2x ISP (13 MP + 13 MP or 25 MP) @ 30 fps

STM32U585 Arm® Cortex®-M33 32-bit MCU

The UNO Q integrates the STM32U585 microcontroller featuring:

- Arm® Cortex®-M33 up to 160 MHz
- 2 MB flash memory
- 786 kB SRAM
- Floating Point Unit