We thank all the reviewers for their constructive comments. Below are detailed responses.

R1&R3: Co-design process elaboration. We provide a simple pseudo-code in Alg. A due to space limit. We will provide details in the final draft.

Algorithm A. The co-design process.

```
# TinyNAS: sample a DNN arch
for arch in arch_space:
  # TinyEngine: find a good schedule
  for schedule in schedule_space:
    # check if satisfy mem. constraints
    if can_fit_memory(arch, schedule):
      # eval acc. and update best arch
      acc = get_valid_acc(arch)
      best_acc = max(best_acc, acc)
      break
```

R1: More deployment devices and tasks. MCUNet generalizes well across different MCU devices with different capacities: we show the ImageNet top-1 accuracy on F746 (320kB SRAM, 1MB Flash) and H743 (512kB SRAM, 2MB Flash) in Table A, MCUNet consistently outperforms the baseline by a large margin (up to 20.4%). MCUNet also generalizes beyond classification to detection. On PASCAL VOC with YOLO, MCUNet significantly improves the mAP from 31.6% to 51.4% on H743. To the best of our knowledge, this is the first large-scale object detection experiment on tiny MCU devices.

	ImgNet(F746)	ImgNet(H743)	VOC(H743)
MbV2+CMSIS	39.7%	53.8%	31.6%
MCUNet	**60.1%**	**65.1%**	**51.4%**

Table A. MCUNet shows consistent improvement across different devices (F746, H743) and tasks (classification, detection).

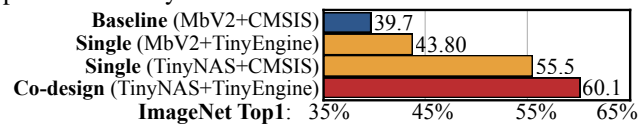

Figure A. MCUNet's co-design scheme outperforms single-design ones on ImageNet classification.

R1: Improvements from co-design over single-design. We showed the advantage of the co-design scheme in Table 2 of the original paper, where co-design achieves 4.6% higher accuracy compared to the best single-design result. We highlight the advantage of the co-design scheme in Figure A. We will make it more clear in the final draft.

R1: Whether the overall network topology brings major improvement. Yes, considering the overall network topology enables specialized im2col, specialized loop tiling and unrolling strategies, which accounts for 49% of the overall performance boost achieved by TinyEngine.

R2: Why the auto-tuning in TVM fails to work on MCUs. MicroTVM's auto-tuning is based on a pre-defined implementation template. However, the template does not include our advanced optimizations, *e.g.*, scheduling memory according to the overall network topology. Therefore, auto-tuning cannot match our speedup and memory reduction.

R4: Contributions of TinyNAS. We would like to clarify that TinyNAS is novel for the "actual NAS procedure". TinyML on MCU is a very new area; existing NAS methods *cannot* fit the tight memory constraints. TinyNAS is the *first* NAS algorithm to enable large-scale deep learning on MCU devices. Since there is no carefully-tweaked design space like those for mobile phones, we have to start from a huge search space so that it is likely to contain a good model for various MCUs. The space needs to cover not only the micro-level architecture designs (*e.g.*, kernel size, expansion ratio) but also the macro-level designs like input resolution and channel widths (Section 3.1). Existing NAS methods fail to achieve good performance on the huge space (Table 5 in original paper), since the large space makes weight-sharing difficult and leads to *low* sample efficiency due to the *sparse* search reward. Our TinyNAS overcomes the search inefficiency with a two-stage search algorithm. The first stage is to shrink/prune the huge search space to a smaller sub-space, so the reward is no longer sparse, and the sample efficiency is improved. The second stage is to perform micro-level optimization in the pruned sub-space. Both stages are the "actual NAS procedure"; they work jointly with TinyEngine to achieve a decent performance, and should not be considered separately.

R4: Comparison to budget-aware NAS methods. TinyNAS argues that a two-stage algorithm that gradually narrows down the search space is important to avoid the sparse search award. Therefore, a fair comparison needs to start from the same full space. We modify existing NAS methods to use the same search space under the same memory constraint as ours. Compared to Single Path One-Shot NAS (SPOS) [17] and Once-For-All (OFA) [5] on ImageNet-100 (ImgN_{100}), TinyNAS outperforms both SOTA methods (Table B), which verifies the advantage of our two-stage search mechanism. Other NAS methods (*e.g.*, [6, 44]) cannot handle the macro-level architecture like backbone channel widths like ours. Therefore, we scale their channels&resolutions to fit the same memory budget of STM32F746 (320kB). Under the same MobileNet-v2 search space, TinyNAS shows significant advantage on ImageNet (ImgN_{1k}) with up to **9.5%** better top-1 accuracy, which verifies memory-awareness is important for TinyML.

Table B. Compare NAS.

Method	ImgN_{100}	ImgN_{1k}
MnasNet [14]	-	51.8%
FBNet [44]	-	50.6%
Proxyless [6]	-	54.4%
SPOS [17]	75.6%	53.6%
OFA [5]	77.0%	54.0%
TinyNAS	**78.7%**	**60.1%**

R4: Existing NAS methods that optimize memory footprint. The two papers provided by the reviewer do not optimize the working memory footprint. MorphNet [Gordon *et al.*, 2018] only considers FLOPs and model size as constraints. Though [Veniat *et al.*, 2018] mentions "memory consumption cost", it actually refers to model size but not activation memory, which is the bottleneck. Neither explored memory-bounded NAS at tiny MCU scale (<1MB).

R4: Details about the experimental protocol. Many experimental protocol details are provided in Section 4.1 and Section G of the supplementary (*e.g.*, datasets, momentum, weight decay, training epochs). We will add more details to the main paper in the final version to help reproduction.

R4: Limited space for NAS. Both stages are the actual NAS procedure to search a good model from a huge search space. Therefore, we have dedicated a considerable amount of space for the NAS procedure. Due to the space limit, we put some of the details of the second stage in the supplementary. We will add it to the main paper in the final version.