We thank the reviewers for their helpful comments. The code and models will be open-sourced.

Reviewer 1

Inference speed test: Along with peak RAM, we report inference FLOPs for all the models. We see a consistent reduction in FLOPs per inference using RNNPool. Resource-constrained devices like microcontrollers, the key focus of our work, do not have parallelism – so the reduction in FLOPs translates reasonably accurately to faster inference; we will add inference numbers on Cortex M-class devices in the manuscript. On hardware with parallelism, the application of RNNPool operator to multiple patches in an activation map can be fully parallelized. For face detection on 320×240 image, the RNNPool layer has about 4800 patches that can be executed in parallel – sufficient for many cloud hardware.

Inappropriate comparison of experiments: The main goal of Table 1 (that includes DenseNet121 on ImageNet-10) is to show that RNNPool works well with a variety of architectures. While we agree that DenseNet121 result might be an artifact of a big model applied to small ImageNet-10 dataset, we observe a similar trend for smaller models like MobileNet also (that are in fact more critical for the resource-constrained settings that we study). We observe a similar trend for ImageNet-1K as well (Table 3) but focus mainly on MobileNets due to resource-constrained devices setting. Table 2 also shows that pooling/downsampling operators significantly affect accuracy in both DenseNet121 and MobileNetV2. Finally, the rationale behind ImageNet-10 can be found in Appendix A.1.

Reviewer 2

FLOPs of ReNet: In Table 2 second block, we compare the accuracy of various downsampling methods while keeping the working RAM same for each. Therefore, ReNet and RNNPool were placed in the same position and have the same shape of input-output feature maps. So when the output map from RNNPool has $4h_2$ channels, the hidden state size of the RNNs in ReNet was taken to be $2h_2$ to ensure an output map of $4h_2$ channels. And even then ReNet's accuracy is significantly worse than RNNPool on multiple tasks. Furthermore, in ReNet each input patch is flattened and passed as a single input to an RNN which leads to a huge matrix multiplication operation. Finally, ReNet authors suggest using GRU or LSTM as the RNN unit; we use GRU as it is more efficient. RNNPool uses FastGRNN as the RNN unit which also contributes to the reduction in FLOPs.

Similarity to ReNet: We respectfully disagree with the reviewer. The global and local contexts do not depend on overlapping or non-overlapping patches but are captured by the constituent operations in a layer. As mentioned in L106-123, ReNet applies each horizontal and vertical RNN sweep on the *entire* image in *one pass* and the resulting hidden states provide the output feature map. Even if ReNet uses overlapping patches, it will still apply both RNNs in *one pass* over the image. In contrast, RNNPool is applied *patchwise* (it is never applied to the entire image in one pass) and only the *final* states from each RNN run are used for output feature map. So, the intuition and goal of the two methods seem quite different. We do present a comparison with ReNet (seen as a downsampling operator) in Table 2 on two different tasks and with three different base models across 2 datasets.

Comparison to DNN compression methods: RNNPool is a complementary technique to model compression, i.e., we can use model compression (e.g. pruning, quantization) on RNNPool based models as well to get additional reduction in working RAM, model size & inference FLOPs. To illustrate this, we created the RNNPool-Face-Quant (Table 4) model using a standard quantization based compression method that maintains accuracy with about $4\times$ compression.

Reviewer 3

Quantization methods: See response to R2; we will add references to the manuscript. As mentioned above, RNNPool is complementary to and supports quantization (and other standard model compression techniques).

Training from scratch: RNNPool is a new building block (pooling operator with parameters) for CNNs. As is the case with any architecture involving a new building block, training the model end-to-end provides the best performance. XNOR-Nets are also trained from scratch because of the formulation and could be coupled with RNNPool based models if needed during end-to-end training.

Training of RNNPool & underlying difficulty: The instances of RNNPool used in our models typically require a sequence length of 8 for the RNNs, and thus are reasonably stable to train. We use the same hyperparameters as the base CNN models to train the RNNPool variants. We do observe that the RNNPool based variants require more epochs to train than the base CNN models. But, overall, the training complexity of RNNPool based models is similar to the base CNN models. We will add training accuracy vs epochs plot in the appendix.

Reviewer 4

Thank you for the positive feedback. The pointer to SqueezeNAS and making a stronger case for RNNPool with the argument is helpful. We will include them in the next revision and fix the citation to "Seeing AI".

Deployment on a microcontroller: We will open-source the code to deploy RNNPool based models on ARM Cortex M-class microcontrollers, and add some of the details in the appendix.