RNNPool: Efficient Non-linear Pooling for RAM Constrained Inference

Oindrila Saha[†], **Aditya Kusupati**[‡],
Harsha Vardhan Simhadri[†], **Manik Varma**[†] **and Prateek Jain**[†]
[†]Microsoft Research India, [‡]University of Washington
{t-oisaha,harshasi,manik,prajain}@microsoft.com, kusupati@cs.washington.edu

Abstract

Standard Convolutional Neural Networks (CNNs) designed for computer vision tasks tend to have large intermediate activation maps. These require large working memory and are thus unsuitable for deployment on resource-constrained devices typically used for inference on the edge. Aggressively downsampling the images via pooling or strided convolutions can address the problem but leads to a significant decrease in accuracy due to gross aggregation of the feature map by standard pooling operators. In this paper, we introduce RNNPool, a novel pooling operator based on Recurrent Neural Networks (RNNs), that efficiently aggregates features over large patches of an image and rapidly downsamples activation maps. Empirical evaluation indicates that an RNNPool layer can effectively replace multiple blocks in a variety of architectures such as MobileNets, DenseNet when applied to standard vision tasks like image classification and face detection. That is, RNNPool can significantly decrease computational complexity and peak memory usage for inference while retaining comparable accuracy. We use RNNPool with the standard S3FD [50] architecture to construct a face detection method that achieves state-of-the-art MAP for tiny ARM Cortex-M4 class microcontrollers with under 256 KB of RAM. Code is released at https://github.com/Microsoft/EdgeML.

1 Introduction

Convolutional Neural Networks (CNNs) have become ubiquitous for computer vision tasks such as image classification and face detection. Steady progress has led to new CNN architectures that are increasingly accurate, but also require larger memory and more computation for inference. The increased inference complexity renders these models unsuitable for resource-constrained processors that are commonplace on the edge in IoT systems and battery-powered and privacy-centric devices.

To reduce inference complexity, several techniques like quantization [44], sparsification [9, 27], cheaper CNN blocks [37, 22], or neural architecture search [41] have been proposed to train CNN models with lower inference cost and model size while retaining accuracy. However, these models still require large working memory for inference. Memory tends to be the most constrained resource on low power devices as it occupies a large fraction of the device die and has high sustained power requirement [24]. Most low power ARM Cortex-M* microcontrollers have less than 256 KB RAM.

Typical CNNs have large intermediate activation maps, as well as many convolution layers, which put together require large amount of RAM for inference (see Proposition 1). A standard approach to reducing working memory is to use pooling operators or strided convolution to bring down size of the activation map. In fact, standard CNNs have multiple such layers. However, such pooling operators aggregate the underlying activation map in a simplistic manner, which can lead to a significant loss of accuracy. As a result, their use is limited to small receptive fields, typically no larger than 3×3, and they can not be used to aggressively reduce the activation map by aggregating larger receptive fields.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

In this paper, we propose a novel pooling operator RNNPool that uses Recurrent Neural Networks (RNNs) to perform a more refined aggregation over a large receptive field of the activation map without compromising on accuracy. RNNPool can be applied to any tensor structured problem, but we focus on 2D images for ease of exposition. For images, RNNPool uses RNNs to aggregate information along rows & columns in a given patch. RNNPool has three parameters – patch size or receptive field, stride, and output dimension – to control its expressiveness and ability to downsample. The RNNPool operator matches standard pooling operators syntactically, so can be used to replace them in convolutional networks.

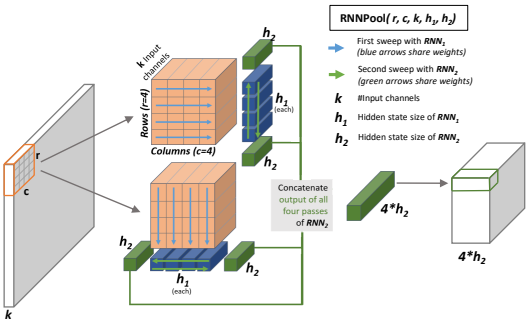

Figure 1: The RNNPool operator applied to patches of size $r \times c$ with stride s. It summarizes the patch with two RNNs into a vector of size $4h_2$.

RNNPool allows rapid down-sampling of images and activation maps, eliminating the need for many memory-intensive intermediate layers. RNNPool is most effective when used to replace multiple CNN blocks in the initial stages of the network where the activation map sizes are large, and hence, require the most memory and compute. There, a single layer of RNNPool can down-sample by a factor of 4 or 8. For example, RNNPool applied to a $640 \times 640 \times 3$ image with patch-size 16, stride 8, and 32 output channels results in a $80 \times 80 \times 32$ activation map, which can be stored in about 200 KB, and can be computed one patch at a time without significant memory cost. Replacing a few blocks using RNNPool reduces peak memory requirement significantly for typical CNN architectures without much loss of accuracy.

Our experiments demonstrate that RNNPool can be used as an effective replacement for multi-layered, expensive CNN blocks in a variety of architectures such as MobileNets, DenseNets, S3FD, and for varied tasks such as image classification and face detection. For example, in a 10-class image classification task, RNNPool+MobileNetV2 reduces the peak memory requirement of MobileNetV2 by up to $10\times$ and MAdds (MAdds refers to Multiply-Adds as in MobileNetV2 [37]) by about 25%, while maintaining the *same* accuracy. Additionally, due to its general formulation, RNNPool can replace pooling layers anywhere in the architecture. For example, it can replace the final average pool layer in MobileNetV2 and improve accuracy by $\sim 1\%$.

Finally, we modify the S3FD [50] architecture with RNNPool to construct an accurate face detection model which needs only 225 KB RAM – small enough to be deployed on a Cortex-M4 based device – and achieves 0.78 MAP on the medium category of the WIDER FACE dataset [47] using $80\times$ fewer MAdds than EXTD [48] – a state-of-the-art resource-constrained face detection method.

In summary, we make the following contributions:

- A novel pooling operator that can rapidly down-sample input in a variety of standard CNN architectures, e.g., MobileNetV2, DenseNet121 while retaining the expressiveness.
- Demonstrate that RNNPool can reduce working memory and compute requirements for image classification and Visual Wake Words significantly while retaining comparable accuracy.
- By combining RNNPool with S3FD, we obtain a state-of-the-art face detection model for ARM Cortex-M4 class devices.

2 Related Work

Pooling: Max-pooling, average-pooling and strided convolution layers [29] are standard techniques for feature aggregation and for reducing spatial resolution in CNNs. Existing literature on rethinking pooling [51, 15, 10] focuses mainly on increasing accuracy and does not take compute/memory efficiency into consideration which is the primary focus of this paper.

Efficient CNN architectures: Most existing research on efficient CNN architectures aims at reducing model size and number of operations per inference. These methods include designing new architectures such as DenseNet [21], MobileNets [20, 37] or searching for them (ProxylessNAS [3], EfficientNets [41], SqueezeNAS [38]). These architectures do not primarily optimize for the peak working memory, which is a critical constraint on devices powered by tiny microcontrollers. Previ-

ous work on memory-optimized inference manipulates existing convolution operator by reordering computations [5, 28] or performing them in place [13]. However, most of these methods provide relatively small memory savings and are validated on low-resolution images like CIFAR-10 [25]. Channel pruning [17] is a method that tries to reduce memory requirement by pruning out multiple convolution kernels in every layer. While effective, channel/filter pruning does not tackle gradual spatial downsampling and thus is a complementary technique to RNNPool.

Visual Wake Words: Visual cues (visual wake word) to "wake-up" AI-powered home assistant devices require real-time inference on relatively small devices. Chowdhery et al. [6] proposed a Visual Wake Words dataset and a resource-constrained setting to evaluate various methods. Section 5.2 discusses the efficient RNNPool based models and their performance for this task.

Face-detection on tiny devices: Recent work including EXTD [48], LFFD [18], FaceBoxes [49] and EagleEye [52] address the problem of accurate real-time face detection on resource-constrained devices. EXTD and LFFD are the most accurate but have high compute and memory requirements. On the other hand, EagleEye and FaceBoxes have lower inference complexity but also suffer from lower MAP scores. Face detection using RNNPool is discussed in Section 5.3.

RNNs for Computer Vision: RNNs have been successful for sequential tasks but haven't been extensively explored in the context of computer vision. An early work, ReNet [42], uses RNN based layer as a replacement for a convolution layer but does not aim at improving efficiency. RNNPool contrasts with ReNet as follows:

a. ReNet is designed to replace a convolutional layer by capturing the global context and leaves the local context to be captured by flattening non-overlapping patches. RNNPool, on the other hand, uses overlapping patches and strongly captures local features and relies on subsequent standard convolutions to capture the global context. Hence, RNNPool and ReNet are complementary methods and can be combined.
b. Semantically, RNNPool is a generalized pooling operator and can replace any pooling layer or strided convolution. However, ReNet does not correspond to any pooling abstraction, making it hard to combine with existing CNN models. For example, RNNPool can modify S3FD architecture to achieve state-of-the-art real-time face detection with < 1 MB RAM while ReNet fails to fit in that context as a replacement layer since the receptive field of the output of ReNet layer varies across spatial positions.
c. ReNet can still be used as a rapid downsampling layer. Table 2 shows that RNNPool outperforms ReNet with lower model size and fewer MAdds across datasets and architectures. E.g. ReNet+MobileNetV2 applied to ImageNet-1K is almost 4% less accurate than RNNPool +MobileNetV2, despite the same working RAM requirement and more MAdds per inference.

Inside-Outside Net [2] uses a ReNet based layer for extracting context features in object detection while PiCANet [31] uses it as a global attention function for salient object detection. L-RNN [45] inserts multiple ReNet based layers but in a cascading fashion. See Appendix B for more discussion.

PolygonRNN [1], CNN-RNN [43] and Conv-LSTM [46] also use RNNs in their architectures but only to model certain sequences in the respective tasks rather than tackling pooling and efficiency.

3 What is RNNPool?

Consider the output of an intermediate layer in a CNN of size $R \times C \times f$, where R and C are the number of rows and columns and f is the number of channels. A typical 2×2 pooling layer (e.g. max or average) with stride 2 would halve the number of rows and columns. So, reducing dimensions by a factor of 4 would require *two* such blocks of convolutions and pooling. Our goal is to reduce the activation of size $R \times C \times f$ to, say, $R/4 \times C/4 \times f'$ or smaller in a single layer while retaining the information necessary for the downstream task. We do so using an RNNPoolLayer illustrated in Figure 1 that utilizes strided RNNPool operators.

3.1 The RNNPool Operator and the RNNPoolLayer

An RNNPool operator of size (r, c, k, h_1, h_2) takes as input an activation patch of size $r \times c \times k$ corresponding to k input channels, and uses a pair of RNNs – RNN_1 of hidden dimension h_1 and RNN_2 with hidden dimension h_2 – to sweep the patch horizontally and vertically to produce a summary of size $1 \times 1 \times 4h_2$.

Algorithm 1 describes the RNNPool operator wich applies two parallel pipelines to a patch and concatenates their outputs. In the first, RNN_1 traverses each row and summarizes the patch horizontally (Line 12) and then RNN_2 trverses the outputs of RNN_1 (Lines 13-14) bi-directionally. In the second pipeline RNN_1 first traverses along columns to summarize the patch vertically (Line 15) and then RNN_2 (Lines 16-17) summarizes bi-directionally.

While it is possible to use GRU [4] or LSTM [19] for the two instances of RNN in RNNPool, we use FastGRNN [26] for its compact size and fewer MAdds (see Appendix H).

An RNNPoolLayer consists of a single RNNPool operator strided over an input activation map and takes as input two more parameters: patch size and the stride length. Note that there are only two RNNs (RNN_1 & RNN_2) in an RNNPool operator, thus weights are shared for both the row-wise and column-wise passes (RNN_1) and all bi-directional passes (RNN_2) across every instance of RNNPool in an RNNPoolLayer.

Algorithm 1 RNNPool Operation

Input: $\mathbf{X} : [\mathbf{x}_{1,1} \dots \mathbf{x}_{r,c}]; \mathbf{x}_{i,j} \in \mathcal{R}^k$
Output: $\text{RNNPool}(\mathbf{X})$

1: **function** $\text{FastGRNN}(\mathcal{P}, \mathbf{x})$
2: $\quad[\mathbf{W}, \mathbf{U}, \mathbf{b}_z, \mathbf{b}_h] \leftarrow \mathcal{P}$, $\mathbf{h}_0 \leftarrow \text{randn}$
3: $\quad$**for** $k \leftarrow 1$ to $\text{length}(\mathbf{x})$ **do**
4: $\quad\quad\mathbf{z} \leftarrow \sigma(\mathbf{W}\mathbf{x}_k + \mathbf{U}\mathbf{h}_{k-1} + \mathbf{b}_z)$
5: $\quad\quad\tilde{\mathbf{h}}_k \leftarrow \tanh(\mathbf{W}\mathbf{x}_k + \mathbf{U}\mathbf{h}_{k-1} + \mathbf{b}_h)$
6: $\quad\quad\mathbf{h}_k \leftarrow \mathbf{z} \odot \mathbf{h}_{k-1} + (\mathbf{1} - \mathbf{z}) \odot \tilde{\mathbf{h}}_k$
7: $\quad$**end for**
8: $\quad$**return** $\mathbf{h}_T$
9: **end function**

10: $\text{RNN}_i(\_) \leftarrow \text{FastGRNN}(\mathcal{P}_i, \_)$, for $i \in \{1, 2\}$
11: **function** $\text{RNNPool}(\mathbf{X})$
12: $\quad\mathbf{p}_i^r \leftarrow \text{RNN}_1(\mathbf{X}_{i,1 \le j \le c})$, for all $1 \le i \le r$
13: $\quad\mathbf{q}^{r_1} \leftarrow \text{RNN}_2(\mathbf{p}^r_{1 \le i \le r})$
14: $\quad\tilde{\mathbf{p}}^r \leftarrow \text{reverse}(\mathbf{p}^r)$, $\mathbf{q}^{r_2} \leftarrow \text{RNN}_2(\tilde{\mathbf{p}}^r_{1 \le i \le r})$
15: $\quad\mathbf{p}_j^c \leftarrow \text{RNN}_1(\mathbf{X}_{1 \le i \le r, j})$, for all $1 \le j \le c$
16: $\quad\mathbf{q}^{c_1} \leftarrow \text{RNN}_2(\mathbf{p}^c_{1 \le j \le c})$
17: $\quad\tilde{\mathbf{p}}^c \leftarrow \text{reverse}(\mathbf{p}^c)$, $\mathbf{q}^{c_2} \leftarrow \text{RNN}_2(\tilde{\mathbf{p}}^c_{1 \le j \le c})$
18: $\quad$**return** $[\mathbf{q}^{r_1}, \mathbf{q}^{r_2}, \mathbf{q}^{c_1}, \mathbf{q}^{c_2}]$
19: **end function**

3.2 Probing the Efficacy of RNNPool

Capturing edges, orientations, and shapes: To demonstrate the capabilities of RNNs as spatial operators for vision tasks such as capturing edges, orientations, and shapes, we performed experiments on synthetic data. RNNPool learns how to capture edges, orientations, and shapes as effectively as convolutional layers which reinforces the choice of RNNs as spatial operators. Appendix C.1 provides further details of these experiments.

Comparing performance with pooling operators: We also performed experiments to contrast the down-sampling power of RNNPool against standard pooling operators on CIFAR-10 [25]. As discussed in Appendix C.2, RNNPool significantly outperforms standard pooling operators in terms of accuracy.

4 How to use the RNNPoolLayer?

RNNPool can be used to modify standard CNN architectures and reduce their working memory as well as computational requirements. Typically, such modifications involve replacing one or more stacks of convolutions and pooling layers of the "base" (original) architecture with an RNNPoolLayer and retraining from scratch. We describe architecture modification strategies here and demonstrate their effectiveness through extensive experimentation in Section 5.

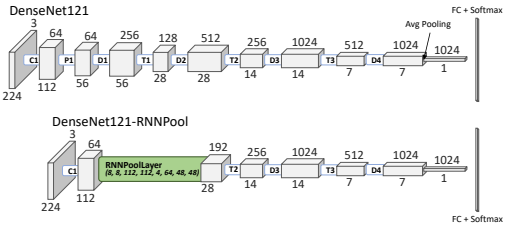

Figure 2: **DenseNet121-**RNNPool: obtained by replacing P1, D1, T1 and D2 blocks in DenseNet121 with an RNNPoolLayer.

Replacement for a Sequence of Blocks: Consider the DenseNet121 [21] architecture in Figure 2. It consists of one convolutional layer, followed by repetitions of "Dense" (D), "Transition" (T) and "Pooling" (P) blocks which gradually reduce the size of the image while increasing the number of channels. Of all these layers, the first block following the initial convolutional layer (D1) requires the most working memory and compute as it works on large activation maps that are yet to be down-sampled. Further, the presence of 6 layers within each dense block makes it harder to work with small memory (see Proposition 1). This is also true of other architectures such as MobileNetV2, EfficientNet, and ResNet.

Table 1: Comparison of inference complexity and accuracy with and without RNNPoolLayer on ImageNet-10.

Model	Base						RNNPool			
	Accuracy (%)	Parameters	Memory Optimised		Standard Calculation [6, 37]					
			Peak RAM	MAdds	Peak RAM	MAdds	Accuracy (%)	Parameters	Peak RAM	MAdds
MobileNetV2	94.20	2.20M	0.38 MB	1.00G	2.29 MB	0.30G	**94.40**	**2.00M**	**0.24 MB**	**0.23G**
EfficientNet-B0	96.00	4.03M	0.40 MB	1.09G	2.29 MB	0.39G	**96.40**	**3.90M**	**0.25 MB**	**0.33G**
ResNet18	**94.80**	11.20M	0.38 MB	21.58G	3.06 MB	1.80G	94.40	**10.60M**	**0.38 MB**	**0.95G**
DenseNet121	**95.40**	6.96M	1.53 MB	24.41G	3.06 MB	2.83G	94.80	**5.60M**	**0.77 MB**	**1.04G**
GoogLeNet	**96.00**	9.96M	1.63 MB	3.32G	3.06 MB	1.57G	95.60	**9.35M**	**0.78 MB**	**0.81G**

We can use an RNNPoolLayer to rapidly down-sample the image size and bypass intermediate large spatial resolution activations. In DenseNet121, we can replace 4 blocks - P1, D1, T1, D2 - spanning *39* layers with a single RNNPoolLayer to reduce the activation map from size $112 \times 112 \times 64$ to $28 \times 28 \times 128$ (see Figure 2). The replacement RNNPoolLayer can be executed patch-by-patch without re-computation, thus reducing the need to store the entire activation map across the image. These two factors greatly reduce the working memory size as well as the number of computations. DenseNet121-RNNPool achieves an accuracy of 94.8% on ImageNet-10 (see Appendix A for dataset details) which is comparable to 95.4% of the original DenseNet121 model.

A similar replacement of functional blocks with RNNPoolLayer can be performed for MobileNetV2 as specified in Table 10 of Appendix F, and leads to a similar reduction in the size of the largest activation map while retaining accuracy. These results extend to other networks like EfficientNet, ResNet and GoogLeNet [40], where residual connection based functional blocks in the initial parts can be effectively replaced with the RNNPoolLayer with improvements in working memory and compute, while retaining comparable accuracy. These results are listed in Table 1. Appendix H presents further ablation studies on RNNPool and its base model.

Replacement for Pooling Layers: RNNPool has the same input and output interface as any pooling operator and hence, RNNPoolLayer can replace any standard pooling layer while providing more accurate aggregation. For example, DenseNet121-RNNPool has three pooling layers one each in T2, T3, and the final average pool layer. Table 1 shows that, on ImageNet-10, DensetNet121-RNNPool loses 0.6% accuracy compared to its base model. But, replacing all three remaining pooling layers in DenseNet121-RNNPool with a RNNPoolLayer results in almost the same accuracy as the *base* DenseNet121 but with about $2\times$ and $4\times$ lower compute and RAM requirement respectively. We can further drop 14 dense layers in D3 and 10 layers in D4 to bring down MAdds and RAM requirement to 0.79G MAdds and 0.43 MB, respectively, while still ensuring 94.2% accuracy.

Replacement in Face Detection models: As in the above architectures, we can use RNNPoolLayer to rapidly down-sample the image by a factor of 4×4 in the early phase of an S3FD face detector [50]. The resulting set of architectures (with different parameters) are described in Appendix F.2. For example, the RNNPool-Face-Quant model has a state-of-the-art MAP for methods that are constrained to at most 256 KB of working RAM (Table 4).

Inference memory requirements: Computing exact memory and compute requirement of a large CNN model is challenging as the execution order of activations in various layers can be re-organized to trade-off memory and compute. For example, in the *memory-optimized* column of Table 1 we present the compute usage of a variety of baseline architectures when their execution order (EO) is restricted to using no more memory than the corresponding RNNPool based architecture. That is, we identify the memory bottleneck layers in various architectures whose activation map size is almost same as that of the corresponding RNNPool-based model. We then compute every voxel of this layer by re-computing the required set of convolutions, *without* storing them. CNNs, in general, have significant compute requirement and such re-compute intensive optimizations make the architecture infeasible even for large devices, e.g. DenseNet121 requires 24.41G MAdds in this scheme (Table 1).

A standard approach is to restrict execution orders that do not require any re-computation of intermediate activation maps. A straightforward and standard EO is the one where the computation is executed *layer-by-layer* [6, 37]. The memory requirement of such a scheme would correspond to the largest activation map in the architecture, except the output of 1x1 convolution layers which can be computed on the fly. This approach mimics the memory requirement of existing platforms like TF-lite [11] and is proposed as a standard benchmark for comparing resource-constrained inference methods [6]. Following this prior convention, we list the inference complexity for various architectures under the *compute-optimized* columns in Table 1, unless the operation is easy to compute on the fly like 1x1

Table 2: Impact of various downsampling and pooling operators on the accuracy, inference complexity and the model size of three *base* architectures: MobileNetV2 and DenseNet121 for ImageNet-10 dataset, and MobileNetV2-0.35x for Visual Wake Word dataset. First block of the table represents the base network and a modified network where the last average pooling layer in the network is replaced by RNNPoolLayer. Second block represent modified networks where the image is passed through a convolution layer followed by various downsampling methods to reduce the size of image by a factor of 4×4. The last row represents the architecture from the second block with RNNPoolLayer with an additional RNNPool replacing the last layer. Peak RAM usage computed using standard convention of [6] is the same for all methods in the second block. Note that RNNPoolLayer +Last layer RNNPool has accuracy similar to the base network while other methods like ReNet are 2-3% less accurate.

Method	**ImageNet-10**						**Visual Wake Words**		
	MobileNetV2			DenseNet121			MobileNetV2-0.35×		
	Accuracy (%)	MAdds	Parameters	Accuracy (%)	MAdds	Parameters	Accuracy (%)	MAdds	Parameters
Base Network	94.20	0.300G	2.2M	**95.40**	2.83G	6.96M	90.20	53.2M	296K
Last layer RNNPool	95.00	0.334G	2.9M	**95.40**	3.05G	7.41M	**91.14**	53.4M	300K
Average Pooling	90.80	**0.200G**	**2.0M**	92.80	**0.71G**	**5.59M**	86.85	**31.9M**	**255K**
Max Pooling	92.80	**0.200G**	**2.0M**	93.40	**0.71G**	**5.59M**	86.92	**31.9M**	**255K**
Strided Convolution	93.00	0.258G	2.1M	93.80	1.33G	6.38M	88.08	39.2M	264K
ReNet	92.20	0.296G	2.3M	93.00	1.35G	6.41M	88.10	46.4M	277K
RNNPoolLayer	94.40	0.226G	**2.0M**	94.80	1.04G	5.60M	89.57	37.7M	**255K**
RNNPoolLayer + Last layer RNNPool	**95.60**	0.260G	2.7M	95.00	1.26G	6.06M	89.65	37.9M	259K

convolution or patch-by-patch computation of RNNPool. Appendix E.2 provides more details about these calculations.

The above scheme is easy to implement and allows an inference pipeline that is more modular and easy to debug and could allow faster inference on neural network accelerators [23]. But, in principle, one can design execution orders (EO) that do not re-compute any intermediate layers, but are still not required to store entire activation maps, especially the largest ones. So, a rigorous quantification of the memory requirement of a model (without any re-compute) needs to show that any valid execution order requires a certain amount of working memory at some point in its execution, and also demonstrate a valid EO with the same memory requirement as a matching upper bound. We achieve this with the following proposition, whose proof and corollaries are in Appendix D.

Proposition 1 *Consider an l-layer ($l > 1$) convolutional network with a final layer of size $m \times n$. Suppose the for each node in the output layer, the size of receptive field in intermediate layer $q \in [l-1]$ is $(2k_q+1) \times (2k_q+1), k_q > 0$ and that this layer has c_q channels and stride 1. Any serial execution order of this network that disallows re-computation requires at least $2\sum_{q=1}^{l-1} c_q k_q \times min(m-1, n-1)$ memory for nodes in the intermediate layers.*

The above proposition shows that for a CNN with receptive field k_q at the q-th layer, the memory requirement scales linearly with the height/width of the activation map and with the number of layers. As networks like MobileNetV2 or DenseNet have blocks with a significant number of convolution layers and large receptive field, this proposition implies that it is not possible to significantly reduce the memory requirement over the standard *layer-by-layer* approach. For example, our un-optimized calculations for RNNPool architectures still give us $3-4$x reduction in peak RAM usage when compared to the *minimum* RAM requirement of the corresponding base architecture (see Appendix E.1). Further, similar optimization can be applied to RNNPool based architectures, so the relative reduction in memory by RNNPool does not change significantly. The implications of the above proposition, i.e., the peak memory of various networks without re-compute is calculated in Appendix E.1.

5 Evaluation of RNNPool on Vision Tasks

We present empirical evidence that RNNPool operator is compatible with popular CNN architectures for vision tasks, and can push the envelope of compute/memory usage vs accuracy curve. Further, we show that RNNPool combined with MobileNetV2 [37] generates accurate models for Visual wake words and face detection problems that can be deployed on tiny Cortex-M4 microcontrollers. See Appendix G for more details about model training and hyperparameters used for the experiments.

5.1 RNNPool for Image Classification

We first focus on ImageNet-10, a 10 class subset of ImageNet-1K [7] where the classes correspond to the categories in CIFAR-10 [25]. We study this dataset because in several realistic tiny devices scenario, like intrusion detection, we are interested in identifying the presence/absence of a few, rather than 1000, classes of objects. The dataset is divided into 1300 images for training and 50 for validation per class. More details and rationale about the dataset can be found in the Appendix A.

Table 2 compares RNNPoolLayer against other standard pooling operators as used in MobileNetV2 and DenseNet121 base networks (see Appendix F.1 for description of the architecture). It shows that with the same memory usage, RNNPool is up to 4% more accurate than the standard pooling operators. While standard pooling operators are cheaper than RNNPool, the overall compute requirement of RNNPool based architectures is similar to pooling based architectures. Furthermore, replacing the last average pooling layer in the base network with RNNPool further increases accuracy, thus demonstrating the flexibility of RNNPoolLayer. Table 2 also contrasts RNNPool with ReNet [42] as a downsampling layer. We observe that RNNPool is a much better alternative for downsampling layers in terms of accuracy (better by up to 2%), model size, and MAdds for the same amount of working memory.

Next, we study the compatibility of RNNPool with different architectures. Table 1 shows that RNNPool based architectures maintain the accuracy of base models while significantly decreasing memory and compute requirement. See Section 4 and Appendix E for a discussion on the calculation of memory and compute requirements of different models.

Table 3: Comparison of resources and accuracy with MobileNets for ImageNet-1K.

Method	Peak RAM	Parameters	MAdds	Accuracy (%)
MobileNetV1	3.06MB	4.2M	569M	**69.52**
MobileNetV1-ReNet	0.77MB	4.2M	487M	66.90
MobileNetV1-RNNPool	**0.77MB**	**4.1M**	**417M**	69.39
MobileNetV2	2.29MB	3.4M	300M	**71.81**
MobileNetV2-ReNet	0.24MB	3.6M	296M	66.72
MobileNetV2-RNNPool	**0.24MB**	**3.2M**	**226M**	70.14

Finally, Table 3 presents results on the complete ImageNet-1K [7] dataset with MobileNetV1 and MobileNetV2 as the base architectures. ReNet and RNNPool based models are constructed in a manner similar to the models in Table 1. See Table 10 for the complete specification of the MobileNetV2+RNNPool model. MobileNetV1+RNNPool model is constructed similarly with $h_1 = h_2 = 16$. Consistent with the results on ImageNet-10, RNNPool retains almost same accuracy as the base models while decreasing memory usage significantly. Furthermore, RNNPool based models are also $3 - 4\%$ more accurate than *ReNet based models*. In this work, we focus on state-of-the-art resource-constrained models that do not require neural architecture search (NAS); we leave extension of RNNPool for NAS based architectures like EfficientNets [41] for future work.

5.2 RNNPool for Visual Wake Words

The Visual Wake Words challenge [6] presents a relevant use case for computer vision on tiny microcontrollers. It requires detecting the presence of a human in the frame with very little resources — no more than 250 KB peak RAM usage and model size, and no more than 60M MAdds/image. The existing state-of-the-art method [6] is MobileNetV2-$0.35\times$ with 8 channels for the first convolution and 320 channels for the last convolution layer. We use this as our baseline and replace convolutions with an RNNPoolLayer. After training a floating-point model with the best validation accuracy, we perform per-channel quantization to obtain 8-bit integer weights and activations.

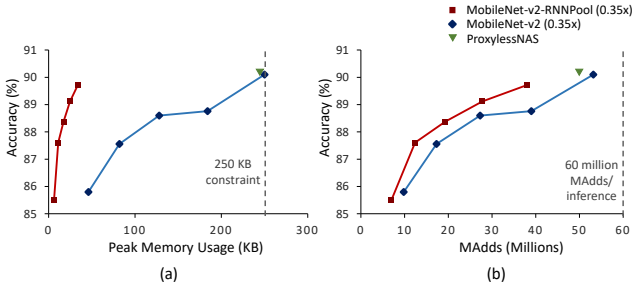

Figure 3: Visual Wake Word: MobileNetV2-RNNPool requires $8\times$ less RAM and 40% less compute than baselines. We cap the number of parameters at $\leq$ 250K instead of the 290K allowed by MobileNetV2 ($0.35\times$). ProxylessNAS has 242K parameters.

Table 2 compares the accuracy of the baseline and new architectures on this task. Replacing the last average pool layer with RNNPool increases the accuracy by $\geq 1\%$. Inserting RNNPool both at the beginning of the network and at the end provides a model whose accuracy is within 0.6% of the

Table 4: Comparison of memory requirement, no. of parameters and validation MAP of various Face Detection architectures when applied to 640×480 RGB images from the Wider Face dataset. RNNPool-Face-C achieves higher accuracy than the baselines despite using $3\times$ less RAM and $4.5\times$ less MAdds. RNNPool-Face-Quant enables deployment on Cortex-M4 class devices with 6-7% accuracy gains over the cheapest baselines.

Method	Peak RAM	Parameters	MAdds	MAP			MAP for ≤ 3 faces		
				Easy	Medium	Hard	Easy	Medium	Hard
EXTD	18.75 MB	**0.07M**	8.49G	0.90	0.88	**0.82**	0.93	0.93	0.91
LFFD	18.75 MB	2.15M	9.25G	0.91	0.88	0.77	0.83	0.83	0.82
RNNPool-Face-C	**6.44 MB**	1.52M	**1.80G**	**0.92**	**0.89**	0.70	**0.95**	**0.94**	**0.92**
FaceBoxes	1.76 MB	**1.01M**	2.84G	0.84	0.77	0.39	-	-	-
RNNPool-Face-B	1.76 MB	1.12M	**1.18G**	**0.87**	**0.84**	**0.67**	0.91	0.90	0.88
EagleEye	1.17 MB	0.23M	**0.08G**	0.74	0.70	0.44	0.79	0.78	0.75
RNNPool-Face-A	1.17 MB	**0.06M**	0.10G	0.77	0.75	0.53	0.81	0.79	0.77
RNNPool-Face-Quant	**225 KB**	0.07M	0.12G	**0.80**	**0.78**	**0.53**	**0.84**	**0.83**	**0.81**

baseline but with far smaller memory requirement (250 $\rightarrow$ 33.68 KB), model size, and MAdds. Peak memory usage is calculated using the same convention as [6].

Further, we sweep across input image resolutions of {96, 128, 160, 192, 224} to trade-off between accuracy and efficiency. Figure 3 shows that RNNPool models are significantly cheaper during inference in terms of compute and memory while offering the same accuracy as the baselines. For example, peak memory usage of MobileNetV2-0.35$\times$ with the lowest resolution images is $\sim$40 KB, while our model requires only 34 KB RAM despite using the highest resolution image and providing $\sim$4% higher accuracy. Note that ProxylessNAS [14] was the winner of the Visual Wake Words challenge. We report it's accuracy on the final network provided by the authors. To be consistent, we train the model only on the training data provided, instead of pretraining with ImageNet-1K used by ProxylessNAS in the wake word challenge.

5.3 RNNPool for Face Detection

We experiment with multiple architectures we call RNNPool-Face-* for face detection suggested in Section 4 and described in greater detail in Appendix F.2. We train and validate these architectures with the WIDER FACE dataset [47]. Versions Quant, A, B, and C of the RNNPool-Face use RNNPoolLayer of hidden dimensions 4, 4, 6 and 16, respectively.

Table 4 compares validation Mean Average Precision (MAP) for easy, medium, and hard subsets. MAP is a standard metric for face detection and measures the mean area under the precision-recall curve. We report MAP scores for baselines based on the official open-source code or pre-trained models. For Eagle-Eye [52], we re-implemented the method as the source code was not available. For EXTD [48], we report MAdds of the EXTD-32 version - the computationally cheapest. EXTD and LFFD [18] are accurate but are computationally expensive. In contrast, RNNPool-Face-C achieves better MAP in the easy and medium subsets despite using $\sim 4.5\times$ less compute and $\sim 3\times$ less RAM.

FaceBoxes [49] and Eagle-Eye reduce MAdds and peak memory usage by aggressively down-sampling the image or by decreasing the number of channels leading to inaccurate models. In contrast, RNNPool-Face-A and RNNPool-Face-B achieve significantly higher MAPs than these methods while still ensuring smaller MAdds and peak RAM usage. We also compare MAP scores for images that have ≤ 3 faces, which is a more realistic face-detection setting for tiny devices. Here also, RNNPool-Face-C is more accurate than all the baselines. Finally, RNNPool-Face-Quant uses byte quantization to reduce the model size so it can be deployed on Cortex-M4 devices which typically have $\leq$ 256 KB RAM, while still having $>$ 0.80 MAP accuracy on images with ≤ 3 faces. See Appendix I for a qualitative evaluation of our method against the baselines.

5.4 RNNPool based Model for ARM Cortex-M4 Microcontrollers

Finally, we develop a face detection model for conference/class room settings that can be deployed on ARM Cortex-M4 class devices. To this end, we develop a more compact version of the face detection model, RNNPool-Face-M4 (Table 15 in Appendix F.2), which has only 4 MBConv blocks. For

Table 5: Comparison of resources and MAP on the SCUT-HEAD dataset. RNNPool-Face-M4 can be effectively deployed on an M4 device with <256 KB RAM in contrast to MobileNetV2-SSDLite low-cost detection model.

Model	MAP	Peak RAM	MAdds	Model Size
MobileNetV2-SSDLite	**0.63**	3.51 MB	540M	11.32 MB
RNNPool-Face-M4	0.58	**188 KB**	**70M**	**160 KB**

further reduction in MAdds and model-size, we train the RNNPool parameters to be sparse. That is, $\mathbf{W}$ matrix of RNN_1 is 50% non-zeros while the rest of the matrices in RNNPool are 30% non-zeros.

To not overshoot RAM for storing input image, we use $320\times240\times1$ monochrome images for training and testing. For evaluation, we first train on the WIDER FACE dataset and then fine-tune on the SCUT-HEAD dataset [35] which consists of images in conference/class rooms. We then use the SeeDot [12] compiler to quantize our model to 8 bits and generate C code for deployment. Table 5 compares the resource requirements and MAP on the SCUT-HEAD validation set (random 80%-20% split) of RNNPool-Face-M4 against a similarly trained MobileNetV2-SSDLite model which is a state-of-the-art architecture for low-cost detection.

Note that MobileNetV2-SSDLite cannot be deployed on a Cortex-M4 device even with 8-bit quantization as the peak RAM requirement is much more than the 256 KB limit of the device. RNNPool-Face-M4 model processes a single image in 10.45 seconds on an ARM Cortex-M4 microcontroller based STM32F439-M4 device clocked at 168 MHz.

6 Conclusions

In this paper, we proposed RNNPool, an efficient RNN-based pooling operator that can be used to rapidly downsample activation map sizes thus significantly reduce inference-time memory and compute requirements for a variety of standard CNNs. Due to syntax level similarity with pooling layers, we can use RNNPool in most existing CNN based architectures. These replacements retain accuracy for tasks like image classification and visual wake words. Our S3FD based RNNPool model for face detection provided accurate models that can be deployed on tiny Cortex-M4 microcontrollers. Finally, we showed with Proposition 1 that calculations of minimum memory requirement for standard architectures can be made rigorous and demonstrate that despite such optimizations of standard CNNs, RNNPool based models can be significantly more efficient in terms of inference-time working memory. Using neural architecture search for RNNPool based models to further reduce inference cost is an immediate and interesting direction.

Broader Impact

Pros: ML models are compute-intensive and are typically served on power-intensive cloud hardware with a large resource footprint that adds to the global energy footprint. Our models can help reduce this footprint by (a) allowing low power edge sensors with small memory to analyze images and admit only interesting images for cloud inference, and (b) reducing the inference complexity of the cloud models themselves. Further, edge-first inference enabled by our work can reduce reliance on networks and also help provide privacy guarantees to end-user. Furthermore, vision models on tiny edge devices enables accessible technologies, e.g., Seeing AI [33] for people with visual impairment.

Cons: While our intentions are to enable socially valuable use cases, this technology can enable cheap, low-latency and low-power tracking systems that could enable intrusive surveillance by malicious actors. Similarly, abuse of technology in certain wearables is also possible.

Again, we emphasize that it depends on the user to see the adaptation to either of these scenarios.

Acknowledgements

We are grateful to Shikhar Jaiswal and Aayan Kumar for their assistance in the deployment of RNNPool models on Cortex-M4 devices. We also thank Sahil Bhatia, Ali Farhadi, Sachin Goyal, Max Horton, Sham Kakade and Ajay Manchepalli for helpful discussions and feedback. Aditya Kusupati did a part of this work during his research fellowship at Microsoft Research India.

References

[1] D. Acuna, H. Ling, A. Kar, and S. Fidler. Efficient interactive annotation of segmentation datasets with polygon-rnn++. In *The IEEE conference on Computer Vision and Pattern Recognition*, pages 859–868, 2018.

[2] S. Bell, C. Lawrence Zitnick, K. Bala, and R. Girshick. Inside-outside net: Detecting objects in context with skip pooling and recurrent neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition*, June 2016.

[3] H. Cai, L. Zhu, and S. Han. ProxylessNAS: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[4] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[5] M. Cho and D. Brand. Mec: memory-efficient convolution for deep neural network. In *International Conference on Machine Learning*, pages 815–824. JMLR. org, 2017.

[6] A. Chowdhery, P. Warden, J. Shlens, A. Howard, and R. Rhodes. Visual wake words dataset. *arXiv preprint arXiv:1906.05721*, 2019.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *The IEEE conference on Computer Vision and Pattern Recognition*, pages 248–255. Ieee, 2009.

[8] D. K. Dennis, Y. Gaurkar, S. Gopinath, S. Goyal, C. Gupta, M. Jain, S. Jaiswal, A. Kumar, A. Kusupati, C. Lovett, S. G. Patil, O. Saha, and H. V. Simhadri. EdgeML: Machine Learning for resource-constrained edge devices. URL https://github.com/Microsoft/EdgeML.

[9] T. Gale, E. Elsen, and S. Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

[10] Y. Gong, L. Wang, R. Guo, and S. Lazebnik. Multi-scale orderless pooling of deep convolutional activation features. In *European Conference on Computer Vision*, pages 392–407. Springer, 2014.

[11] Google. ML for mobile and edge devices - tensorflow lite. URL https://www.tensorflow.org/lite.

[12] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma. Compiling kb-sized machine learning models to tiny iot devices. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–95, 2019.

[13] A. Gural and B. Murmann. Memory-optimal direct convolutions for maximizing classification accuracy in embedded applications. In *International Conference on Machine Learning*, pages 2515–2524, 2019.

[14] S. Han, J. Lin, K. Wang, T. Wang, and Z. Wu. Solution to Visual Wakeup Words Challenge'19 (first place). URL https://github.com/mit-han-lab/VWW.

[15] K. He, X. Zhang, S. Ren, and J. Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *The IEEE transactions on Pattern Analysis and Machine Intelligence*, 37 (9):1904–1916, 2015.

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *The IEEE conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[17] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *The IEEE International Conference on Computer Vision*, pages 1389–1397, 2017.

[18] Y. He, D. Xu, L. Wu, M. Jian, S. Xiang, and C. Pan. LFFD: A light and fast face detector for edge devices. *arXiv preprint arXiv:1904.10633*, 2019.

[19] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

[20] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[21] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *The IEEE conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017.

[22] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[23] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture*, pages 1–12, 2017.

[24] D. Kim, J.-Y. Choi, and J.-E. Hong. Evaluating energy efficiency of internet of things software architecture based on reusable software components. *International Journal of Distributed Sensor Networks*, 13(1):1550147716682738, 2017.

[25] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[26] A. Kusupati, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma. FastGRNN: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network. In *Advances in Neural Information Processing Systems*, pages 9017–9028, 2018.

[27] A. Kusupati, V. Ramanujan, R. Somani, M. Wortsman, P. Jain, S. Kakade, and A. Farhadi. Soft threshold weight reparameterization for learnable sparsity. In *International Conference on Machine Learning*, 2020.

[28] L. Lai, N. Suda, and V. Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601*, 2018.

[29] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[30] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision*, pages 740–755. Springer, 2014.

[31] N. Liu, J. Han, and M.-H. Yang. Picanet: Learning pixel-wise contextual attention for saliency detection. In *The IEEE Conference on Computer Vision and Pattern Recognition*, June 2018.

[32] A. Mead. Review of the development of multidimensional scaling methods. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 41(1):27–39, 1992.

[33] Microsoft. Seeing AI. URL https://www.microsoft.com/en-us/ai/seeing-ai.

[34] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[35] D. Peng, Z. Sun, Z. Chen, Z. Cai, L. Xie, and L. Jin. Detecting heads using feature refine net and cascaded multi-scale architecture. *arXiv preprint arXiv:1803.09256*, 2018.

[36] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[37] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *The IEEE conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[38] A. Shaw, D. Hunter, F. Iandola, and S. Sidhu. SqueezeNAS: Fast neural architecture search for faster semantic segmentation. In *ICCV Neural Architects Workshop*, 2019.

[39] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning*, pages 1139–1147, 2013.

[40] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *The IEEE conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[41] M. Tan and Q. Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114, 2019.

[42] F. Visin, K. Kastner, K. Cho, M. Matteucci, A. Courville, and Y. Bengio. Renet: A recurrent neural network based alternative to convolutional networks. *arXiv preprint arXiv:1505.00393*, 2015.

[43] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu. CNN-RNN: A unified framework for multi-label image classification. In *The IEEE conference on Computer Vision and Pattern Recognition*, pages 2285–2294, 2016.

[44] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han. Haq: Hardware-aware automated quantization with mixed precision. In *The IEEE conference on Computer Vision and Pattern Recognition*, pages 8612–8620, 2019.

[45] W. Xie, A. Noble, and A. Zisserman. Layer recurrent neural networks. 2016.

[46] S. Xingjian, Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, and W.-c. Woo. Convolutional LSTM network: A machine learning approach for precipitation nowcasting. In *Advances in Neural Information Processing Systems*, pages 802–810, 2015.

[47] S. Yang, P. Luo, C.-C. Loy, and X. Tang. Wider face: A face detection benchmark. In *The IEEE conference on Computer Vision and Pattern Recognition*, pages 5525–5533, 2016.

[48] Y. Yoo, D. Han, and S. Yun. EXTD: Extremely tiny face detector via iterative filter reuse. *arXiv preprint arXiv:1906.06579*, 2019.

[49] S. Zhang, X. Zhu, Z. Lei, H. Shi, X. Wang, and S. Z. Li. Faceboxes: A CPU real-time face detector with high accuracy. In *The IEEE International Joint Conference on Biometrics*, pages 1–9. IEEE, 2017.

[50] S. Zhang, X. Zhu, Z. Lei, H. Shi, X. Wang, and S. Z. Li. S3fd: Single shot scale-invariant face detector. In *The IEEE International Conference on Computer Vision*, pages 192–201, 2017.

[51] Q. Zhao, S. Lyu, B. Zhang, and W. Feng. Multiactivation pooling method in convolutional neural networks for image recognition. *Wireless Communications and Mobile Computing*, 2018, 2018.

[52] X. Zhao, X. Liang, C. Zhao, M. Tang, and J. Wang. Real-time multi-scale face detector on embedded devices. *Sensors*, 19(9):2158, 2019.

A Dataset Information

A.1 ImageNet-10

The ImageNet-10 is a subset of images from ILSVRC 2012 ImageNet-1K dataset [36] of 1000 classes. All images corresponding to the 10 classes from CIFAR-10 as listed in Table 6 are sampled from the full dataset. The classes in CIFAR-10 are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck.

The class n02430045: 'deer' is not present in the ImageNet-1K subset and was scraped from the full ImageNet-22K database [7]. Each class is divided into 1300 images for training and 50 images for validation.

Table 6: Classes in ImageNet-10 dataset.

Class no.	ImageNet id	Class name
1	n02690373	'airliner'
2	n04285008	'sports car'
3	n01560419	'bulbul'
4	n02124075	'Egyptian cat'
5	n02430045	'deer'
6	n02099601	'golden retriever'
7	n01641577	'bullfrog'
8	n03538406	'horse cart'
9	n03673027	'ocean liner'
10	n04467665	'trailer truck'

Typical on-device models for real-world applications deal with limited classes (e.g. intruder detection). ImageNet-10 is a good proxy for this task with medium resolution natural images.

A.2 Visual Wake Words

This is a binary classification dataset [6] dealing with the presence and absence of a person in the image. The dataset is derived by re-labeling the images available in the MS COCO dataset [30] with labels corresponding to whether a person is present or not. The training set has 115K images and the validation set has 8K images. The labels are balanced between the two classes: 47% of the images in the training dataset of 115k images are labeled as 'person'.

A.3 WIDER FACE

This is a face detection dataset [47] with 32,203 images containing 393,703 labeled faces varying in scale, pose, and occlusion. It is organized based on 61 event classes. Each event class has 40%/10%/50% data as training, validation, and testing sets. The images in the dataset are divided into Easy, Medium, and Hard cases. The Hard case includes all the images of the dataset, and the Easy and Medium cases are subsets of the Hard case. The hard case includes images with a large number of faces or tiny faces along with the data from Easy and Medium cases.

A.4 SCUT HEAD

This is a head detection dataset [35]. We use PartB of this dataset for our experiments. PartB includes 2405 images with 43940 heads annotated. 1905 images of PartB are for training and 500 for testing.

B RNN as a spatial operator and comparison with ReNet

Since ReNet [42], there have been a few methods that have been built upon it to solve various vision tasks. The fundamental difference, mathematically, between these approaches, and ours is how the RNN is used to extract spatial information. In ReNet based methods, the RNN is used to find a pixel-wise mapping from a voxel of the input activation map to that of the output map. However, in our method, we are using RNNs to spatially summarize a big patch of the input activation map to a 1×1 voxel of the output activation map. Note that in ReNet the hidden states of every timestep of RNN contribute to one voxel of the output, whereas in our case only the last hidden states of the traversals are taken for both row/column-wise summarizations and bidirectional summarizations.

ReNet based approaches either insert RNN based layers in existing networks or replace a single convolution layer (thus resulting in increasing computations). In ReNet, the RNNs are applied over the whole input map, whereas RNNPool is applied patch by patch, which is semantically similar to a pooling operator. Our usage of RNN for spatial information extraction is so powerful that we can eliminate a large amount of RAM and compute heavy convolution layers and still preserve accuracy.

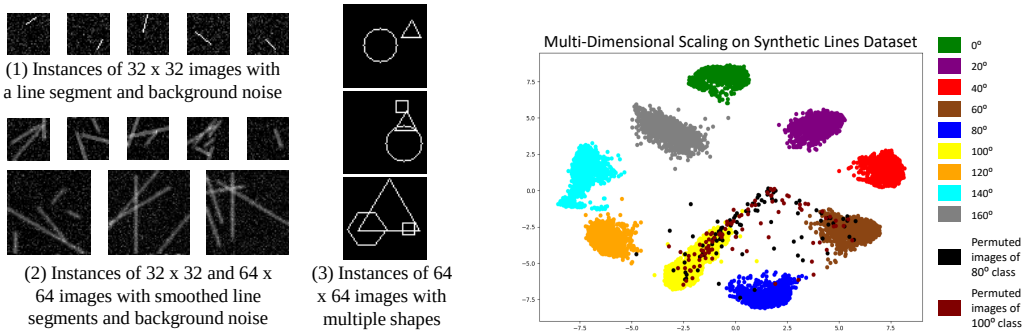

Figure 4: (left) Examples from three multi-class and multi-label synthetic datasets used for probing RNNPool. (right) A 2-dimensional Multi-Dimensional Scaling visualization of the 128 dimensional output of RNNPool operator for the multi-class dataset (1). Some test images (plotted using black and brown dots) were modified by randomly permuting rows and columns.

For ReNet to do the same, patches of size equal to the stride have to be flattened to construct an input to the RNN, which makes it further inefficient in terms of compute and parameters and results in loss of spatial dependencies. RNNPool results in a decrease in computations and parameters while ReNet based methods will increase the same with respect to the baseline model. The comparisons in Table 2 & 3 show that ReNet in fact results in a significant loss in accuracy too.

C Probing the Efficacy of RNNPool

C.1 Capturing Edges, Orientations and Shapes

To probe RNNPool's efficacy at capturing edges, orientation, and shapes, we attempt to fit an RNNPool operator to the following synthetic datasets of small 8-bit monochrome images with background noise as shown in Figure 4. We conduct experiments on synthetic datasets to prove that RNNPoolLayer can learn spatial representations.

1. A multi-class dataset consisting of images with one line segment of varying lengths and positions. There are 9 classes corresponding to lines ranging from 0 to 160° at 20° intervals.
2. A multi-label dataset with images consisting of multizple line segments with varying lengths and positions. There are 9 labels corresponding to lines with orientations of 0 to 160° at 20° intervals.
3. A multi-label dataset consisting of images with a subset of shapes (5 in total) – circle, triangle, square, pentagon, and hexagon.

We sweep over the h_1, h_2 parameters in powers of 2 for the smallest RNNPool operator that can enable a single FC layer to classify or label the test set with 100% accuracy. We do so with and without a preceding CNN layer of 8 convolutions of 3×3 size and stride 2. Table 7 lists the least h_1, h_2 required for each task. We observe that a single RNNPool module fits to 100% accuracy on all these datasets.

Table 7: Minimum required hyperparameter configurations for synthetic experiments.

Data	Image Size	With Conv.	Without Conv.
(1)	32×32	$h_1 = 4, h_2 = 16$	$h_1 = 16, h_2 = 32$
(2)	32×32	$h_1 = h_2 = 8$	$h_1 = h_2 = 32$
(2)	64×64	$h_1 = 8, h_2 = 16$	$h_1 = h_2 = 32$
(3)	64×64	$h_1 = 8 = h_2 = 16$	$h_1 = h_2 = 32$

We conclude that the horizontal and the vertical passes of the RNN allows a single RNNPool operator to capture the orientation of edges and simple shapes over patches of size up to 64×64. Further, adding a single convolutional layer before the RNNPool layer makes the model much more parameter efficient. In effect, the convolution layer detects gradients in a local 3×3 patch, while the RNNPool detects whether gradients across 3×3 patches aggregate into a target shape.

Further, we use multi-dimensional scaling [32] to visualize the $4 \cdot h_2 = 128$ dimensional output of RNNPool operator on the multi-class dataset (1) in Figure 4 (left). Dataset (1) consists of various lines in the image at a discrete set of angles, and the classification task is to detect the angle of

the line. Some images from the test set of classes 80° and 100° are multiplied with a permutation matrix to randomly permute rows and columns. These resulting images are added to the original test dataset and the output of the RNNPool is plotted in Figure 4 (right). The outputs for each class form well-separated tight clusters indicating RNNPool indeed learns various orientations, while the outputs for the permuted images are scattered across the plot indicating that it is not exploiting certain gross aggregations in the data.

C.2 Comparing Performance with Pooling Operators

We now contrast the down-sampling power of RNNPool against standard pooling operators. That is, we investigate if the pooling operators maintain accuracy for a downstream task even when the pooling receptive field is large. To this end, we consider the image classification task with CIFAR-10 dataset [25] but the pooling operator is required to down-sample the input 32×32 image to a 1×1 voxel in *one go* i.e. both patch size and stride are 32. This is followed by a fully connected (FC) layer. The number of output channels after pooling was ensured to be the same. For Max and Average pooling models, a 1×1 convolution is used to ensure the same output dimension. For this task, RNNPool achieves an accuracy of **70.63%**, while the convolution layer, max pooling, and average pooling's accuracy are 53.13%, 20.04% and 26.53%, respectively. This demonstrates the modeling power of the RNNPool operator over other pooling methods. Table 2 (Rows 2-5) reinforces the same but on bigger image classification datasets.

Details. We use $h_1 = h_2 = 32$ for the RNNPool operator with patch size and stride as 32. For the strided convolution we use a convolution layer of $4 \times h_2 = 128$ filters. For Max and Average pooling first we pool down to $1 \times 1 \times 3$ from input of $32 \times 32 \times 3$ and then use a 1×1 convolution of 128 filters. All the above have the same patch size and stride size and are followed by a fully connected layer projection to 10 from 128.

D Lower bounds on space required for multi-layer networks

We now lower bound the memory requirements of computation of multi-layer convolutional networks when recomputation is not permitted. Suppose we have an l-layer ($l > 1$) convolutional network. Let Y denote the nodes in the final layer which form a grid of size $m \times n$. Suppose that the size of the receptive field of each node in Y in an intermediate layer l is $(2k+1) \times (2k+1), k > 0$ and that $y_{i,j} \in Y$ depends on the activations of nodes $x^{(l)}_{i'j'}, i' \in \{i-k, \ldots, i, \ldots i+k\}, j' \in \{j-k, \ldots, j, \ldots j+k\}$ in the intermediate layer l. Suppose further that the convolution operations have stride 1 and are generic and not separable, i.e., can not in the general case be factored into depth-wise separable operations. An execution of this network "disallows recomputation" if once a node x in an intermediate layer (layers that are neither the input nor output of the network) is computed, all nodes $y \in Y$ that depend on x must be computed before x is evicted from memory.

Claim 1 *Fix column $j \in [n]$. Suppose that nodes $y_{i,j},\ i \in I \subsetneq [m]$ have been completed at some point in an execution order. Then at the same point in the execution order, at least $2k$ contiguous activations $x^{(l)}_{i^*-k+1,j}, x^{(l)}_{i^*-k+2,j}, \ldots x^{(l)}_{i^*+k,j}$ for some $i^* \in [m]$ will need to be saved in memory until another node from column j is computed.*

Proof. Since $I \subsetneq [m]$, there exists index $i^* \in [m] \setminus I$ such that either $i^*+1 \in I$ or $i^*-1 \in I$. Suppose without loss of generality that $i^*-1 \in I$. Then, nodes $x^{(l)}_{i^*-k+1,j}, x^{(l)}_{i^*-k+2,j}, \ldots, x^{(l)}_{i^*+k-1,j}$ must have been loaded into memory. However, $y_{i^*,j}$ also depends on these intermediate nodes, and has not yet been computed. So these $2k$ intermediate nodes must be retained in memory, thus proving the statement. The case where $i^*+1 \in I$ is similar.

With this claim, we are ready to prove Proposition 1.

Proof of Proposition 1. Fix any execution order of the network, and label the nodes in the final layer Y in the order they are evaluated: $(p_1, q_1), (p_2, q_2), \ldots, (p_{mn}, q_{mn})$. That is y_{p_1,q_1} is evaluated before y_{p_2,q_2} and so on. Let

$$I_t = \cup_{\tau=1}^{t} p_\tau, \quad J_t = \cup_{\tau=1}^{t} q_\tau, \quad \text{and} \quad t^* = \min_t \{|I_t| = m \text{ or } |J_t| = n\}.$$

That is, once $y_{p_{t^*},q_{t^*}}$ is executed, either (a) at least one node in each row of the final layer has been executed, or (b) at least one node in each column of the final layer has been executed, and at the moment $y_{p_{t^*-1},q_{t^*-1}}$ is computed, there is an entire row, say r, and an entire column, say c, in the final layer where no nodes have been executed.

Suppose that case (b) holds. Then, at step $t^* - 1$, nodes in $n - 1$ columns $[n] \setminus \{c\}$ have been executed, and in each column, at least one row has not been executed. By Claim 1, each such column would need to have $2k_q$ activations at layer q in memory at this point of execution, and all these nodes are unique (that the nodes required to be in memory by Claim 1 for different columns are non-overlapping). Therefore, at least $2\sum_{q=1}^{l-1} c_q k_q \times (n-1)$ memory is required to hold the necessary nodes in each intermediate layer for this execution.

A similar analysis of case (a) yields a lower bound of $2\sum_{q=1}^{l-1} c_q k_q \times (m-1)$ from which the lemma follows. ■

If convolution operators have a stride larger than 1, then we can similarly state the following claim based on the overlap between the nodes in an intermediate layer that are common dependencies across two consecutive rows/columns of the output.

Claim 2 *Fix column $j \in [n]$. Suppose that nodes $y_{i,j}$, $i \in I \subsetneq [m]$ have been completed at some point in an execution order. Suppose that the stride at layer q is s_q. Restrict s_q to 1 in a layer with 1×1 convolutions, i.e., assume activations are not simply thrown away. Then at the same point in the execution order, at least $k' = 2k + 1 - \Pi_{r=q}^{l} s_r$ contiguous activations $x^{(l)}_{i^*-\lfloor k'/2 \rfloor+1,j}, x^{(l)}_{i^*-k+2,j}, \ldots x^{(l)}_{i^*+\lceil k'/2 \rceil,j}$ for some $i^* \in [m]$ will need to be saved in memory until another node from column j is computed.*

This allows us to restate Proposition 1 in networks where stride is greater than 1.

Proposition 2 *Consider an l-layer ($l > 1$) convolutional network with a final layer of size $m \times n$. Suppose the for each node in the output layer, the size of receptive field in intermediate layer $q \in [l-1]$ is $(2k_q + 1) \times (2k_q + 1), k_q > 0$ and that this layer has c_q channels and stride s_q. Restrict s_q to 1 in a layer with 1×1 convolutions. Suppose that $k'_q = 2k_q + 1 - \Pi_{r=q}^{l-1} s_r$. Any serial execution order of this network that disallows re-computation requires at least $\sum_{q=1}^{l-1} c_q k'_q \times min((\Pi_{r=q}^{l-1} s_r)m - 1, (\Pi_{r=q}^{l-1} s_r)n - 1)$ memory for nodes in the intermediate layers.*

Claim 3 *The lower bound in Proposition 1 is matched by an execution order that computes the network in a row or column-first order, whichever is smaller. That is, execute all the intermediate nodes needed to compute the first row of the output, retain those intermediate nodes required for the calculation of the second row of the output, compute the second row of output, and so on. Let $S_q = \Pi_{r=q}^{l} s_r$, and restrict s_q to 1 in a layer with 1×1 convolutions. This schedule has a memory requirement of $\sum_{q=1}^{l-1} c_q(2k_q + 1 - S_q)min(Q_q m - 1 + 2k_q, S_q n - 1 + 2k_q)$ if we account for the padding at either ends of the row in each intermediate layer, and*

$$\sum_{q=1}^{l-1} c_q(2k_q + 1 - S_q)min(S_q m - 1, S_q n - 1),$$

if the padding is not counted.

Claim 4 *Suppose we follow the row (or column)-wise execution order in Claim 3, and that each row in the output depends on k_0 layers at the input. Suppose that the input is required to be in memory before the start of the execution and the output is required to be in memory at the end of the execution. Let c_{in} and c_{out} denote the number of channels in the input and output. Let $S_q = \Pi_{r=q}^{l} s_r$, and let $k'_0 = k_0 - S_1$ be the number of rows/columns in the input layer that are common dependencies between two consecutive rows/columns of the output. The memory requirement including* ***those of the input and output layers*** *is*

$$\max\{m_{in}n_{in}c_{in} + k'_0 n_{out}c_{out}, m_{out}n_{out}c_{out} + k'_0 n_{in}c_{in}\} + \sum_{q=1}^{l-1} c_q(2k_q + 1 - S_q)min(S_q m - 1, S_q n - 1),$$

with padding added on the fly for convolutions at the boundaries of activation maps. This is obtained by reclaiming the footprint of the input for the output one row at time (with a lag of k_0 rows) once all the nodes that depend on it are completed.

E Details about Compute and Peak RAM Calculation

In this section, we quantify the memory requirements of the networks analyzed in this paper.

E.1 Optimal memory requirements without recomputation

First, we analyze the minimum memory requirements and optimal execution orders of components – inverted residual block, separable residual block, dense block, and inception block – assuming that no re-computation is allowed. That is, we wish to find the minimum value, over all valid execution orders E of the block, of the maximum memory requirement of the execution order. Then, we analyze the memory requirement of image classification architectures discussed in this paper.

E.1.1 Memory requirements of various block

We assume that the execution always starts with the input of the block in memory, and terminates with output in memory. We denote that the size of input I is $h_{in} \times w_{in} \times C$, where h_{in} and w_{in} are the height and the width of the activation and c_{in} is the number of channels. Likewise, denote the size of O to be $h_{out} \times w_{out} \times c_{out}$. In what follows, suppose also that $h_{in} \geq w_{in}$ and $h_{out} \geq w_{out}$. Otherwise we can flip rows and columns and meet the same constraints.

1. **Inverted bottleneck residual block (a.k.a. MBConv, see Fig. 3b of [37])** : The first layer is a point-wise convolution (C1) that expands the number of channels to $c_{in} \times t$ where t is expansion factor. Then there is a depth-wise separable 3×3 convolution (C2) with stride either 1 or 2, followed by another point-wise convolution (C3) which reduces the number of output channels. We can use the row-wise order suggested in Claim 4, which results in a schedule where the first row of the output is generated, then the second row and so on. This schedule has a memory footprint of $\max\{h_{in}w_{in}c_{in} + (3-s)w_{out}c_{out}, h_{out}w_{out}c_{out} + (3-s)w_{in}c_{in}\} + (3-s)tc_{in}w_{in}$, where s is the stride of the 3×3 convolution.
2. **Residual Block (see Fig. 5(left) of [16])** : We consider a residual block consisting of two convolution layers with 3×3 kernels, of which the first has a stride s of 1 or 2, and the second has stride 1. The we have $w_{out} = w_{in}/s$ and $h_{out} = h_{in}/s$. Using Claim 4, we can see that the best case memory footprint is $\max\{h_{in}w_{in}c_{in} + (5-s)w_{in}c_{out}/s, h_{in}w_{in}c_{out}/s^2 + (5-s)w_{in}c_{in}\} + 2w_{in}c_{out}/s$, assuming that the number of channels of intermediate layer is equal to c_{out} as is the norm here.
3. **Inception block (see Fig. 2b of [40])**: Denote the output of each of the 4 paths in the block by O_1, O_2, O_3 and O_4. We consider the case where all convolutions are of stride 1. We can apply the arguments of Section D simultaneously for all four paths with slight modification. We consider a minimal set of contiguous rows at the start of the input – which would be first 5 row in the referenced image as its the largest convolution size – and compute all channels in the first row of the output of all four paths. We then drop the first row of input, materialize the second row of output on all four paths and so on. If we denote by c_{out} the number of output channels of all four networks, then the memory requirement is $\max\{h_{in}w_{in}c_{in} + 4w_{out}c_{out}, h_{out}w_{out}c_{out} + 4w_{in}c_{in}\} + (2c_2 + 4c_3)w_{in}$, where c_2 and c_3 are the number of intermediate channels in O_2 and O_3 respectively.
4. **Dense block (see Fig. 4 of URL)** : At any point in the execution of a dense block, we need to store the input to the dense block and outputs of all previous dense layers, since the last layer needs all the activation maps concatenated as its input. The total activation maps being stored will reach the peak just after the last dense layer. Therefore the peak memory requirement is the output of the dense block.

E.1.2 Memory requirements of image classification networks

We calculate the lowest possible memory requirements of networks using calculations in the previous subsection for individual blocks and the following methodology: find a partitioning of a multi-layer

Table 8: Comparison of accuracy, compute and minimum memory requirement for inference with and without RNNPoolLayer on ImageNet-10. The memory calculations reflect the application of Proposition 2 and Claim 4

.

Model	Base				RNNPool			
	Accuracy (%)	Parameters	Peak RAM	MAdds	Accuracy (%)	Parameters	Peak RAM	MAdds
MobileNetV2	94.20	2.20M	0.84MB	0.30G	**94.40**	**2.00M**	**0.24MB**	**0.23G**
EfficientNet-B0	96.00	4.03M	0.84MB	0.39G	**96.40**	**3.90M**	**0.24MB**	**0.33G**
ResNet18	**94.80**	11.20M	0.81MB	1.80G	94.40	**10.60M**	**0.38MB**	**0.95G**
DenseNet121	**95.40**	6.96M	2.38MB	2.83G	94.80	**5.60M**	**0.77MB**	**1.04G**
GoogLeNet	**96.00**	9.96M	1.01MB	1.57G	95.60	**9.35M**	**0.59MB**	**0.81G**

network into disjoint contiguous sets of layers that minimizes the least memory requirement of the most memory-intensive partition. Using this, we calculate the memory requirements of networks in Table 1 and list the requirements in Table 8. We now discuss the specifics of each network, and in particular, the partition of the layers of the network that requires the maximum memory (and thus lower bonds the memory requirement of a network).

GoogLeNet has a initial convolution layer (C1) of stride 2, followed by a max pooling layer (P1), another convolution layer (C2) of stride 2 and then a max pooling layer (P2). Output of P2 is of size $28 \times 28 \times 192$. Applying Proposition 2 to the set of layers starting with the input image (I) and output of P2 (O), the RAM required is 112 × (11-4) × 64 + 56 × (5-2) × 64 + 56 × (3-2) × 192 added to O and 7 rows of input, is lesser than the requirement for inception (3b). For the inception (3b) block, the input is ($28 \times 28 \times 256$) and the output is of size $14 \times 14 \times 480$. Therefore using Proposition 2, the RAM required is 28 × (7-2) × 32 + 28 × (5-2) × 128 + 28 × (3-2) × 64 + 28 × (3-2) × 480 (the first three terms are intermediate activations of the inception block and have different receptive fields), added to the input size (28 × 28 × 256) + 14 × (7-2) × 480, results in 1.01MB.

DenseNet121 has a 2-strided convolution layer (C1) in the beginning followed by a max pool of stride 2 (P1) and then D1-the first Dense block which has 6 Dense layers. Each Dense layer has 1×1 convolution with 128 output channels followed by a 3×3 convolution with 128 input and 32 output channels. The output of each Dense layer is concatenated to the input to form the input to the next Dense layer which is why the 1×1 convolution in each Dense layer has different input channels. D1 is followed by a 1×1 convolution which reduces channels of activation map to half followed by P2, another Max Pool layer. For determining the peak RAM required, we apply Proposition 2 to the set of layers starting with the output of P1 (I) until the output of P2 (O), so that we can go from 56 × 56 × 64 to 28 × 28 × 128 directly bypassing 56 × 56 × 256 sized O_{D1}. The receptive field of O on I can be calculated to be 14×14. The RAM for intermediate activations will be 56 × (14-2) × 128 + 56 × (12-2) × 32 + 56 × (12-2) × 128 + 56 × (10-2) × 32 + . . . + 56 × (4-2) × 32. The total peak RAM along with I (56× 56 × 64) + 28 × (14-2) × 128, which is 2.38MB.

ResNet18. A similar calculation as above can be done for ResNet18. The architecture consists of a convolution layer (C1) of stride 2 followed by a max pool layer (P1), followed by residual blocks. In this case, let us apply Proposition 2 to the block of layers starting with the input RGB image of size $224 \times 224 \times 3$ (denoted I) until the output of P1 (denoted O). Between I and O we have 2 layers: C1 and P1. Therefore the total RAM requirement will be 112 × (3-2) × 64 added to O (56 × 56 × 64) + 224 × (11-4) × 3, which is 0.81MB.

MobileNetV2 has a convolution layer C1 of stride 2 followed by a MBConv block MB1 which has stride 1. MB1 contributes to the peak memory (2.29MB). Denote by I the input RGB image of size $224 \times 224 \times 3$ and denote by O the output of MB1. The receptive field of O on output of C1 is 3, on output of first layer of MB1 is 3 and after the 1 for the rest two layers of MB1. Therefore, using Proposition 1, the RAM required is 112 × (3-1) × 32 + 112 × (3-1) × 32 added to O (112 × 112 × 16)) + 224 × (7-2) × 3, which is 0.84MB.

EfficientNet-B0 has exactly the same calculation as MobileNetV2 as the first convolution block and first MBConv block are identical.

RNNPool Versions : Similar to GoogLeNet we can also reduce peak RAM of GoogLeNet-RNNPool. Here inception (4e) is the bottleneck. Lets take I as the input to inception (3b)(14 × 14 × 528) and O as the output of the pooling layer after inception (3b). Size of O is 7 × 7 × 832. Therefore using Proposition 1, the RAM required is 14 × (7-2) × 32 + 14 × (5-2) × 160 + 14 × (3-2) × 128 + 14 × (3-2) × 832, added to input (14 × 14 × 528) + 7 × (7-2) × 832, resulting in 0.59MB.

The peak memory requirements of RNNPool versions of ResNet18, DenseNet121, MobileNetV2 and EfficientNet-B0 in Table 1 cannot be reduced further by better schedules as we replace the most memory-intensive blocks and operate patch-by-patch, which is more local and granular that row-by-row schedules used above.

E.2 Memory requirement (without recomputation) estimates according to prior conventions

In this subsection, we follow the scheduling convention of Chowdhery et al. [6] to estimate the memory requirements of individual blocks and networks that use them. Note that the memory requirements listed here can be higher than in Section E.1 as the schedules may not be optimal from memory requirement perspective.

E.2.1 Memory requirements of individual blocks

1. **Inverted bottleneck residual block (a.k.a. MBConv)** : Give input I of size $h_{in} \times w_{in} \times C$, a pointwise convolution (C1) first expands the number of channels to $C \times t$ where t is expansion factor. Then there is a depthwise separable 3×3 convolution (C2) with stride either 1 or 2, followed by another pointwise convolution (C3) which reduces the channel to the number of output channels (O) associated with the MBConv block. To avoid storing the large output (O_{C1}) of C1 and bloating the memory, O_{C1} is constructed channel by channel, so at first 1 filter of the $C \times t$ filters of C1 will be convolved with I, then this single 2D vector will be convolved by C2. Since C2 is depthwise separable and input channels independently contribute to an output channel, we again get a 2D map. This map is convolved with all filters of C3 and we get an output of O number of channels. We keep doing this, going one by one through each filter of C1 and adding to the output of the MBConv block of O channels, to get the final output. Hence, the memory requirement is the size of input added to that of the output of the MBConv block.
2. **Residual Block** : The memory requirement is the maximum of input and output maps of the block. As the residual connection adds the input to the output values can be discarded after being added to the output values being computed.
3. **Inception block**: Denote the input to the inception block I and the outputs of each of the 4 paths in the block O_1, O_2, O_3 and O_4. Since we can get rid of the input I after computing the last output, we can order the computation in increasing order of the number of channels in O_i. Therefore, the peak RAM while computing the full block will be the sum of input added to the sum of the 3 smallest outputs.
4. **Dense block**: A dense block needs to store the input as well as outputs of all previous dense layers since the last layer needs all the activation maps concatenated. The volume activation maps stored will reach the peak just after the last dense layer. Therefore the peak RAM usage is the size of the output of the dense block.

E.2.2 Memory requirements of image classification networks in Table 1

We now use the above results to compute the memory requirements of image classification networks, assuming all computations are in 32-bit floating-point. We assume the layer-by-layer convention of [6] for RAM computation. The peak memory requirement of both MobileNetV2 and EfficientNet-B0 is contributed by the first MBConv block in these architectures. The input map size to the block is $112 \times 112 \times 32$ and the output map size is $112 \times 112 \times 16$, adding up to a peak memory requirement of 2.29MB.

The peak memory requirement of the RNNPool inserted versions is the MBConv block right after the RNNPool replacement. The input size is $28 \times 28 \times 64$ and output size is $14 \times 14 \times 64$ for MobileNetV2-RNNPool, adding up to 0.24MB. The input size is $28 \times 28 \times 64$ and output size is $14 \times 14 \times 80$ for EfficientNetB0-RNNPool, adding up to 0.25MB.

For ResNet18, DenseNet121, and GoogLeNet the maximum memory requirement is to host the activation map just after the first convolution layer which is of size $112 \times 112 \times 64$. For ResNet18-RNNPool, the maximum requirement comes from the residual block just after RNNPool, i.e., the first residual block out of the two of conv4_x. The input to this is of size $28 \times 28 \times 128$ and the output size is $14 \times 14 \times 256$. The maximum of these two is 0.38MB. For DenseNet121-RNNPool, the largest memory requirement comes from the output of D3 (see Figure 2), the size of which

$14 \times 14 \times 1024$ i.e. 0.77MB. For GoogLeNet, the peak requirement comes from the last inception block on the spatial resolution of 14×14 — inception (4e). Here the size of the input is $14 \times 14 \times 528$ and sizes of the 3 smallest outputs are $14 \times 14 \times 128$, $14 \times 14 \times 128$ and $14 \times 14 \times 256$, totaling 0.78MB.

E.2.3 Memory requirement of face detection networks in Table 4 without recomputation

We use convention of considering the largest activation map to be the peak RAM requirement. For EagleEye, FaceBoxes, EXTD and LFFD architectures, the largest activation map is the output of the first convolution, their sizes being $320 \times 240 \times 4$ (=1.17MB), $160 \times 120 \times 24$ (=1.76MB), $320 \times 240 \times 64$ (=18.75MB) and $320 \times 240 \times 64$ (=18.75MB) respectively. For RNNPool-Face-A and RNNPool-Face-B, the largest activation map is the output of the RNNPool, which is $160 \times 120 \times 16$ (=1.17MB) and $160 \times 120 \times 24$ (=1.76MB) respectively. For RNNPool-Face-C and RNNPool-Face-Quant, peak memory requirement is contributed by the MBConv block right after the RNNPool. The input size of this block for RNNPool-Face-C is $160 \times 120 \times 64$ and output size is $160 \times 120 \times 24$, the total being 6.44MB. The input size of this block for RNNPool-Face-Quant is $80 \times 60 \times 32$ and output size is $80 \times 60 \times 16$, the total being 224KB as we quantize to 1 byte unsigned integer.

E.3 Memory requirements of image classification networks in Table 1 with recomputation

As explained in Section E.2.2, the RAM calculations for RNNPool based models revealed that the convolution block after RNNPoolLayer contributes to the peak RAM. Let's denote this block in both the base architecture and RNNPool-based version as ConvBlock-A. In the memory-optimized scheme, we fix the peak RAM of the base model to be that of the convolution block whose RAM usage is a bit more than that of the RNNPool version. We denote by ConvBlock-B the convolution block that lies before ConvBlock-A, and such that there exists no block that lies between this block and ConvBlock-A which has a RAM usage less than that of ConvBlock-A. Note that ConvBlock-B is present only in the base model and not the RNNPool model. Since we fix the peak RAM, we have to reconstruct an activation map (denoted by Activation-A) that comes before ConvBlock-B patch by patch. Note that Activation-A need not necessarily be the activation map just before ConvBlock-B. Activation-A is chosen as the earliest occurring activation map (nearer to the input image) which ensures that there is no intermediate layer or block between it and ConvBlock-B which can contribute to more RAM usage. We do construct Activation-A by loading a patch of the image (one at a time), which is of the size of the receptive field of Activation-A w.r.t. the input image, and feed it forward to get a $1 \times 1 \times channel_{Activation-A}$ voxel of Activation-A. When we load the next patch we have to re-compute some convolution and pooling outputs which come in the overlapping region of the two consecutive patches. We keep doing this until we reconstruct Activation-A completely. The total number of MAdds is the sum of the MAdds of the base network and the extra re-computations in order to compute patch-by-patch.

F Architectures

F.1 Image Classification

F.1.1 RNNPoolLayer in the beginning replacing multiple blocks

Table 9: RNNPool settings for image classification.

Model	Hidden Size	Patch Size
MobileNetV2-RNNPool	$h_1 = h_2 = 16$	6
EfficientNet-B0-RNNPool	$h_1 = h_2 = 16$	6
ResNet18-RNNPool	$h_1 = h_2 = 32$	8
DenseNet121-RNNPool	$h_1 = h_2 = 48$	8
GoogLeNet-RNNPool	$h_1 = h_2 = 32$	8
MobileNetV2-RNNPool (0.35×)	$h_1 = h_2 = 8$	6

As discussed in Figure 2, we can use RNNPoolLayer in the beginning of the architecture to rapidly downsample the image leading to smaller working RAM and compute requirement. Table 9 presents

the hidden state size and patch size used by RNNPoolLayer when applied to various models discussed in Table 1. Note that the last row refers to the model used for Visual Wake Words experiments (Figure 3).

Furthermore, Table 10 presents the exact architecture used by MobileNet-v2-RNNPool(0.35x) architecture applied to the Visual Wakeword problem (Section 5.2).

Table 10: MobileNetV2-RNNPool: RNNPool Block with patch-size 6×6 and hidden sizes $h_1 = h_2 = 16$ is used. The rest of the layers are defined as in [37]. Each line denotes a sequence of layers, repeated n times. The first layer of each bottleneck sequence has stride s and rest use stride 1. Expansion factor t is multiplied to the input channels to change the width. The number of output classes is l.

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d 3×3	1	32	1	2
$112^2 \times 32$	RNNPool Block	1	64	1	4
$28^2 \times 64$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1×1	1	1280	1	1
$7^2 \times 1280$	avgpool 7×7	1	-	1	1
$1 \times 1 \times 1280$	conv2d 1×1	1	l	-	1

F.1.2 RNNPoolLayer replacing Average Pooling at the end

Typical image classification models use average pooling before the final feed-forward layer to produce the class probabilities. As RNNPoolLayer is syntactically equivalent to standard pooling layers, we can use it to perform the pooling in the penultimate layer, replacing the average pool layer. To this end, we use RNNPool operator with $h_1 = h_2 = l/4$ where l is the number of channels in the last activation map before the average pooling layer. Such a replacement does not significantly contribute to the number of parameters and MAdds. In Table 2, Row 2 refers to such a replacement in the base MobilnetV2, DenseNet121, and MobilenetV2-0.35x models, while Row 7 refers to similar replacement in the corresponding RNNPool models. In Figure 3, all RNNPool based architectures use RNNPool both in the beginning layer and in the penultimate layer of the network.

F.1.3 RNNPoolLayer replacing intermediate Pooling layers

These experiments have been tried on DenseNet121 as the base model (Section-4), where we are replacing single max-pooling layers appearing in intermediate positions in the network with RNNPool. Given $r_{in} \times c_{in} \times k_{in}$ size input activation map to the pooling layer, the hidden sizes for RNNPool is taken as $h_1 = h_2 = k_{in}/4$, patch size as 4 and stride as 2. Note that we also further drop dense layers (1×1 convolution followed by 3×3 convolution) in D3 and D4. The number of channels in the output of any dense block is the sum of the number of input channels and output of each dense layer. Hence, reducing the number of dense layers reduces the number of channels of the output activation maps of these dense blocks and hence the input to the pooling layer. However, for the RNNPool the same strategy of $h_1 = h_2 = k_{in}/4$ is followed where k_{in} is lesser now.

F.2 Face Detection

Our detection network builds upon the backbone structure of S3FD [50]. Each RNNPool-Face model is created by placing RNNPool Block directly after the input image or after a strided convolution (RNNPool-Face-Quant). Following the RNNPoolLayer, we apply standard S3FD architecture for detection. Detection layers are placed at strides of 4, 8, 16, 32, 64, and 128, for square anchor boxes of sizes 16, 32, 64, 128, 256, and 512 as in S3FD.

Following S3FD architecture, we fix the required receptive field size of each of the detection layers, which is then used to compute the number of MBConv Blocks or convolution layers after RNNPool and before each detection layer. We also use S3FD's anchor matching strategy and the max-out background label technique.

Table 11: The architecture of RNNPool-Face-C

Input	Operator	t	c	n	s
$640 \times 480 \times 3$	RNNPoolLayer	1	64	1	4
$160 \times 120 \times 64$	bottleneck	6	24	2	1
$160 \times 120 \times 24$	bottleneck	6	32	3	2
$80 \times 60 \times 32$	bottleneck	6	64	4	2
$40 \times 30 \times 64$	bottleneck	6	96	3	2
$20 \times 15 \times 96$	bottleneck	6	160	2	2
$10 \times 7 \times 160$	bottleneck	6	320	1	2

Images are trained on 640×640 images. A multi-task loss is used where cross-entropy loss is used for classification of anchor box and smooth L1 loss is used as regression loss for bounding box coordinate offsets. We use multi-scale testing and Non-Maximal Suppression during inference to determine final bounding boxes.

Table 11 contains the architecture of RNNPool-Face-C. There is a detection layer after every bottleneck stack. The detection layer contains two 3×3 constitutional kernels which predict the class probability (2 outputs per pixel) and bounding box offsets(4 outputs per pixel). The convention followed in the table below is the same as in Table 10. t is the expansion coefficient, c is the number of output channels, n is the number of repetitions of the MBConv[1] layer and s is the stride associated with the first of those stack of layers. RNNPool's hidden state sizes are fixed to be: $h_1 = h_2 = 16$.

Table 12: The architecture of RNNPool-Face-B

Input	Operator	t	c	n	s
$640 \times 480 \times 3$	RNNPoolLayer	1	24	1	4
$160 \times 120 \times 24$	conv2d 3×3	1	24	4	1
$160 \times 120 \times 24$	conv2d 3×3	1	96	1	2
$80 \times 60 \times 96$	conv2d 1×1	1	32	1	1
$80 \times 60 \times 32$	bottleneck	6	32	3	1
$80 \times 60 \times 32$	bottleneck	6	64	3	2
$40 \times 30 \times 64$	bottleneck	6	128	2	2
$20 \times 15 \times 128$	bottleneck	6	160	1	2
$10 \times 7 \times 160$	bottleneck	6	320	1	2

Architecture for RNNPool-Face-B is shown in Table 12. The detection heads are after the second row of the table and then after each stack of bottleneck layers. RNNPool's hidden state sizes are fixed to be: $h_1 = h_2 = 6$.

Architecture for RNNPool-Face-A is shown in Table 13. The detection heads are after the second row of the table and then after each stack of bottleneck layers. RNNPool's hidden state sizes are fixed to be: $h_1 = h_2 = 16$. Depthwise+Pointwise refers to a depthwise separable 3×3 convolution followed by a pointwise 1×1 convolution.

The architecture for RNNPool-Face-Quant is shown in Table 14. The detection heads are after the second row of the table and then after each stack of bottleneck layers. The first detection head has a strided 3×3 convolution to reach a total stride of 4 (following S3FD). RNNPool's hidden state sizes are fixed to be: $h_1 = h_2 = 4$.

[1]We use the terms 'bottleneck', MBConv, and inverted residual interchangeably, they refer to the same block.

Table 13: The architecture of RNNPool-Face-A

Input	Operator	t	c	n	s
$640 \times 480 \times 3$	RNNPoolLayer	1	16	1	4
$160 \times 120 \times 16$	Depthwise+Pointwise	1	16	4	1
$160 \times 120 \times 16$	Depthwise+Pointwise	1	16	1	2
$80 \times 60 \times 16$	bottleneck	1	16	3	1
$80 \times 60 \times 16$	bottleneck	1	24	3	2
$40 \times 30 \times 24$	bottleneck	1	32	2	2
$20 \times 15 \times 32$	bottleneck	2	128	1	2
$10 \times 7 \times 128$	bottleneck	2	160	1	2

Table 14: The architecture of RNNPool-Face-Quant

Input	Operator	t	c	n	s
$640 \times 480 \times 3$	conv2d 3×3	1	4	1	2
$320 \times 240 \times 4$	conv2d 3×3	1	4	1	1
$320 \times 240 \times 4$	RNNPoolLayer	1	32	1	4
$80 \times 60 \times 32$	bottleneck	2	16	4	1
$80 \times 60 \times 16$	bottleneck	2	24	4	2
$40 \times 30 \times 24$	bottleneck	2	32	2	2
$20 \times 15 \times 32$	bottleneck	2	64	1	2
$10 \times 7 \times 64$	bottleneck	2	96	1	2

Table 15: The architecture of RNNPool-Face-M4

Input	Operator	t	c	n	s
$320 \times 240 \times 1$	conv2d 3×3	1	4	1	2
$160 \times 120 \times 4$	RNNPoolLayer	1	64	1	4
$40 \times 30 \times 64$	bottleneck	2	32	1	1
$40 \times 30 \times 32$	bottleneck	2	32	1	1
$40 \times 30 \times 32$	bottleneck	2	64	1	2
$20 \times 15 \times 64$	bottleneck	2	64	1	1

Table 15 shows the RNNPool-Face-M4 architecture for our cheapest model deployed on a M4 device. The model has 4 detection layers after each MBConv Block. RNNPool's hidden state sizes are fixed to be: $h_1 = h_2 = 16$.

The RNNPool models decrease MAdds drastically while maintaining performance. Figure 5, shows the difference we are making. When restricted to the methods with <2G MAdds requirement, our model attains even better MAP (for easy and medium dataset) than the state-of-the-art EXTD and LFFD architectures (which need about 10G MAdds per inference.

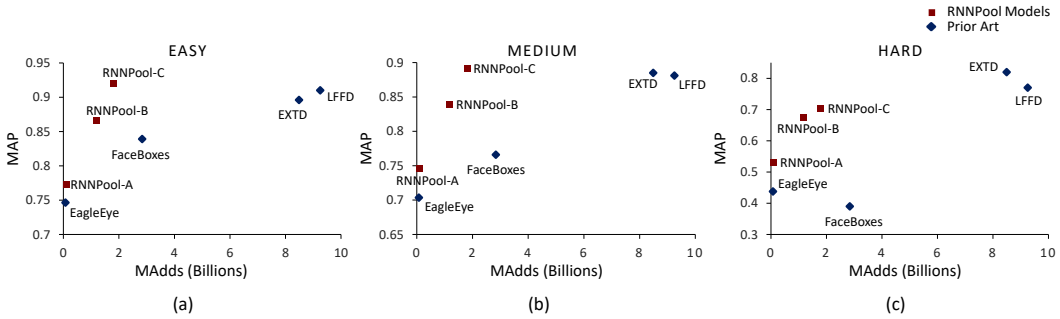

Figure 5: WIDER Face Dataset: MAdds vs MAP of various methods including RNNPool +S3FD.

G Hyperparameters

Models are trained in PyTorch [34] using SGD with momentum optimizer [39] with weight decay 4×10^{-5} and momentum 0.9. We do data-parallel training with 4 NVIDIA P40 GPUs and use a batch size of 256 for classification and 32 for face detection. We use a cosine learning rate schedule with an initial learning rate of 0.05 for classification tasks, and 0.01 with 5 warmup epochs for face detection tasks. All convolution layers use learnable batch normalization. We use the EdgeML [8] implementation of FastGRNN. All ImageNet-10 and face detection experiments were trained for 300 epochs. Both Visual Wake Words and ImageNet-1K experiments were run for 150 epochs. Best top-1 validation accuracy is reported in all the classification datasets and test MAP was reported for face detection.

We use FastGRNN as both the RNNs in RNNPool. We usually use the same hidden dimension for both the RNNs. We fix ζ as 1 and ν as 0 for all models, for stability, and use piecewise linear non-linearities quantTanh and quantSigmoid for the Visual Wake Word models, so we can quantize it without loss of information.

Various image augmentations were used for training each network. For the ImageNet experiments, the training images were cropped to a random size of 0.08 to 1.0 times the original size and reshaped to a random aspect ratio of 3/4 to 4/3. This was then resized to 224×224. This image was further flipped horizontally randomly and then normalized by the mean and standard deviation. For the validation set, we resize the input image to 256×256 and then take a center crop of 224×224. For the Visual Wake Word experiment, we follow a similar process except during training we crop the input image first to a random size of 0.2 to 1.0 times the original size. For varying resolutions from 96 to 224 as reported in Figure 3, the ratio of resizing resolution of the input image and center crop size is kept the same during validation. All other augmentations are kept the same with output size changed from 96 to 224. For Face Detection experiments we use augmentations like in S3FD [50]. This includes color

distortion, random cropping: specifically zooming in to smaller faces to get larger faces to train on, and horizontal flipping after cropping to 640 $\times$ 640. Note that the same augmentation strategies were used for the baseline models also for a fair comparison.

H RNNPool Ablation

In this section, we first discuss the changes in accuracy, peak RAM, MAdds, and the number of parameters on varying hyperparameters of RNNPool like patch size, hidden dimensions, and stride. We also compare the same for multiple layers of RNNPool. We use MobileNetV2 as the base network and the dataset is ImageNet-10. Note that the first row refers to the MobileNetV2-RNNPool architecture in Table 10, and the other rows (b)-(e) of Table 16 are variations on it. Table 16 (f) and (g) have another 4 MBConv blocks replaced in the MobileNetV2-RNNPool architecture (Row 3 of Table 10). (f) uses a single RNNPool to do this replacement whereas (g) uses two consecutive RNNPool Blocks. All variations have $\sim$2M parameters (even (g) which has 2 RNNPool layers has a very minimal model size overhead). This suggests that a finer hyperparameter and architecture search could lead to a better trade-off between accuracy and compute requirements.

Table 16: Comparison of accuracy, peak RAM and MAdds for variations in hidden dimensions, patch size and stride in RNNPool for MobileNetV2 and on ImageNet-10 dataset. Parameters are same as the base if not mentioned. (f) and (g) are further replacements in MobileNetV2-RNNPool (Row 3 of Table 10).

#	Hyperparameters	Accuracy (%)	Peak RAM	MAdds
(a)	Reported (Patch Size = 6; $h_1 = h_2 = 16$, Stride = 4)	94.4	0.24MB	0.23G
(b)	Patch size = 8	94.0	0.24MB	0.24G
(c)	Patch size = 4	93.2	0.24MB	0.22G
(d)	$h_1 = h_2 = 8$	92.8	0.14MB	0.21G
(e)	$h_1 = h_2 = 32$	95.0	0.43MB	0.29G
(f)	Stride = 8; Patch Size = 12	94.0	0.14MB	0.17G
(g)	Stride = 4; Patch Size = 6 and Stride = 2; Patch Size = 4	93.2	0.19MB	0.17G

In Table 18, we ablate over the choice of RNN cell (LSTM, GRU and FastGRNN) in RNNPool for the MobileNetV2-RNNPool model (Table 10) on the ImageNet-10 dataset. We show that the choice of FastGRNN results in significantly lower MAdds than LSTM or GRU while having about 1% higher accuracy. Finally, Table 17 has the training curve for the MobileNetV2-RNNPool on ImagetNet-10 showing that training with RNNPool is not harder than the base models.

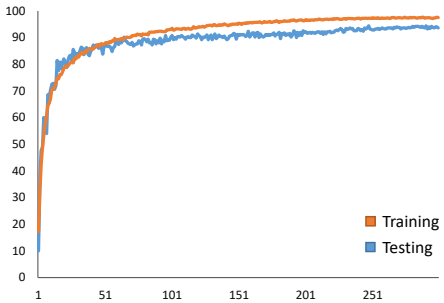

Table 17: Training curve of MobileNetV2-RNNPool on ImageNet-10.

Table 18: Ablation over RNN cell in RNNPool for MobileNetV2-RNNPool on ImageNet-10.

RNN cell	Parameters	MAdds	Accuracy (%)
LSTM	2.0M	266M	93.4
GRU	2.0M	246M	93.0
FastGRNN	**2.0M**	**226M**	**94.4**

I Face Detection Qualitative Results

Figures 6 and 7 show the qualitative results where RNNPool based models outperform the current state-of-the-art real-time face detection models.

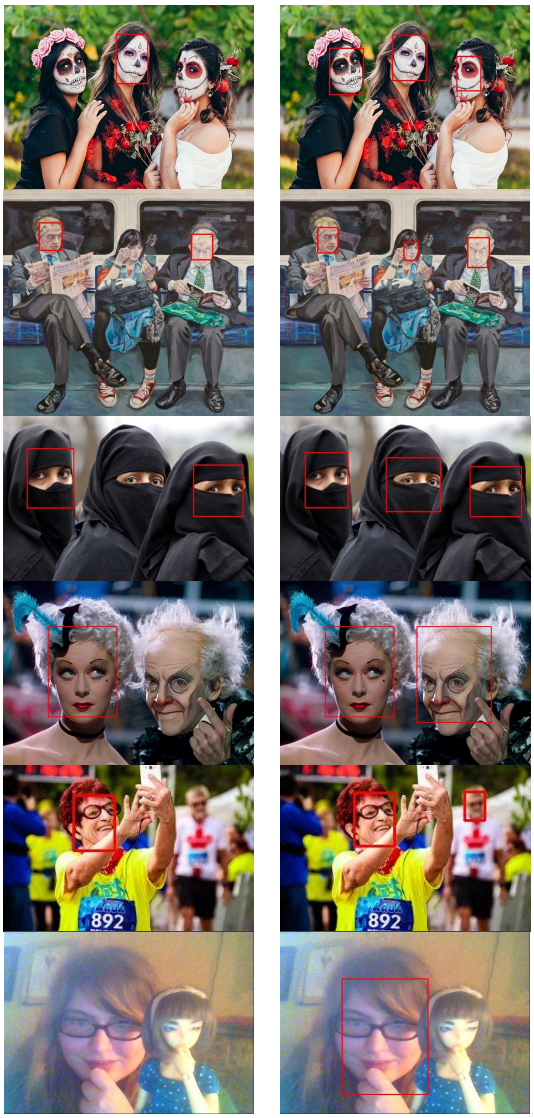

Figure 6: Comparison of performance on test images with Eagle-Eye and RNNPool-Face-Quant. The confidence threshold is set to 0.6 for both models. EagleEye misses faces when there is makeup, occlusion, blurriness and in grainy pictures, while our method detects them. However, in the case of some hard faces, RNNPool-Face-Quant misses a few of them or does not draw a bounding box over the full face.

Figure 7: Comparison of performance on test images with EXTD_32 and RNNPool-Face-C. The confidence threshold is set to 0.6 for both models. The EXTD model has more false positives and misses more faces. In the first image, EXTD makes a faulty prediction at the top right. In the second image, EXTD mistakes regions in leaves for faces, while our model detects just the two correct faces. In the next image, both the models have some wrong detections, but the EXTD model detects a large bounding box that is a false positive. In the next image EXTD misses a face with an unnatural pose that our model detects. However, our model detects a face within a face which in general can be removed easily. In the next image (last row above), both the models detect the two faces, which weren't detected by the models on the left. Our model detects a slightly better bounding box than EXTD.