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

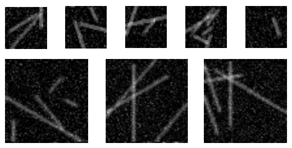
This is a head detection dataset [35]. We use PartB of this dataset for our experiments. PartB includes 2405 images with 43940 heads annotated. 1905 images of PartB are for training and 500 for testing.

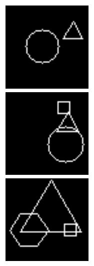
B RNN as a spatial operator and comparison with ReNet

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.

(1) Instances of 32 x 32 images with a line segment and background noise


(2) Instances of 32 x 32 and 64 x 64 images with smoothed line segments and background noise


(3) Instances of 64 x 64 images with multiple shapes

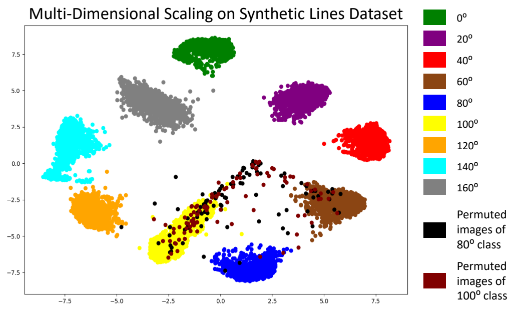


Figure 4: (left) Examples from three multi-class and multi-label synthetic datasets used for probing RNNPool. (right) A 2-dimensional Multi-Dimensional Scaling visualization of the 128 dimensional output of RNNPool operator for the multi-class dataset (1). Some test images (plotted using black and brown 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 $\times$ (11-4) $\times$ 64 + 56 $\times$ (5-2) $\times$ 64 + 56 $\times$ (3-2) $\times$ 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 $\times$ (7-2) $\times$ 32 + 28 $\times$ (5-2) $\times$ 128 + 28 $\times$ (3-2) $\times$ 64 + 28 $\times$ (3-2) $\times$ 480 (the first three terms are intermediate activations of the inception block and have different receptive fields), added to the input size (28 $\times$ 28 $\times$ 256) + 14 $\times$ (7-2) $\times$ 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 $\times$ 1 convolution with 128 output channels followed by a 3 $\times$ 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 $\times$ 1 convolution in each Dense layer has different input channels. D1 is followed by a 1 $\times$ 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 $\times$ 56 $\times$ 64 to 28 $\times$ 28 $\times$ 128 directly bypassing 56 $\times$ 56 $\times$ 256 sized O_{D1}. The receptive field of O on I can be calculated to be 14$\times$14. The RAM for intermediate activations will be 56 $\times$ (14-2) $\times$ 128 + 56 $\times$ (12-2) $\times$ 32 + 56 $\times$ (12-2) $\times$ 128 + 56 $\times$ (10-2) $\times$ 32 + $\ldots$ + 56 $\times$ (4-2) $\times$ 32. The total peak RAM along with I (56$\times$ 56 $\times$ 64) + 28 $\times$ (14-2) $\times$ 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 $\times$ (3-2) $\times$ 64 added to O (56 $\times$ 56 $\times$ 64) + 224 $\times$ (11-4) $\times$ 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 $\times$ (3-1) $\times$ 32 + 112 $\times$ (3-1) $\times$ 32 added to O (112 $\times$ 112 $\times$ 16)) + 224 $\times$ (7-2) $\times$ 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 $\times$ 14 $\times$ 528) and O as the output of the pooling layer after inception (3b). Size of O is 7 $\times$ 7 $\times$ 832. Therefore using Proposition 1, the RAM required is 14 $\times$ (7-2) $\times$ 32 + 14 $\times$ (5-2) $\times$ 160 + 14 $\times$ (3-2) $\times$ 128 + 14 $\times$ (3-2) $\times$ 832, added to input (14 $\times$ 14 $\times$ 528) + 7 $\times$ (7-2) $\times$ 832, resulting in 0.59MB.

The peak memory requirements of $\mathrm{RNNPool}$ 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 $\mathrm{RNNPool}$ inserted versions is the MBConv block right after the $\mathrm{RNNPool}$ replacement. The input size is $28 \times 28 \times 64$ and output size is $14 \times 14 \times 64$ for MobileNetV2-$\mathrm{RNNPool}$, 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-$\mathrm{RNNPool}$, 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-$\mathrm{RNNPool}$, the maximum requirement comes from the residual block just after $\mathrm{RNNPool}$, 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-$\mathrm{RNNPool}$, 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 $\mathrm{RNNPool}$-Face-A and $\mathrm{RNNPool}$-Face-B, the largest activation map is the output of the $\mathrm{RNNPool}$, which is $160 \times 120 \times 16$ (=1.17MB) and $160 \times 120 \times 24$ (=1.76MB) respectively. For $\mathrm{RNNPool}$-Face-C and $\mathrm{RNNPool}$-Face-Quant, peak memory requirement is contributed by the MBConv block right after the $\mathrm{RNNPool}$. The input size of this block for $\mathrm{RNNPool}$-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 $\mathrm{RNNPool}$-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 $\mathrm{RNNPool}$ based models revealed that the convolution block after $\mathrm{RNNPoolLayer}$ contributes to the peak RAM. Let's denote this block in both the base architecture and $\mathrm{RNNPool}$-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 $\mathrm{RNNPool}$ 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 $\mathrm{RNNPool}$ 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 $\mathrm{RNNPoolLayer}$ in the beginning replacing multiple blocks

Table 9: $\mathrm{RNNPool}$ 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 $\mathrm{RNNPoolLayer}$ 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.

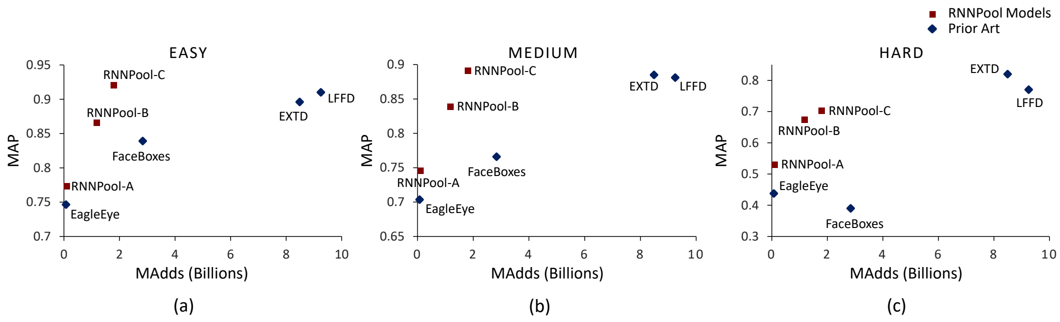


Figure 5: WIDER Face Dataset: MAdds vs MAP of various methods including RNNPool +S3FD.

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.

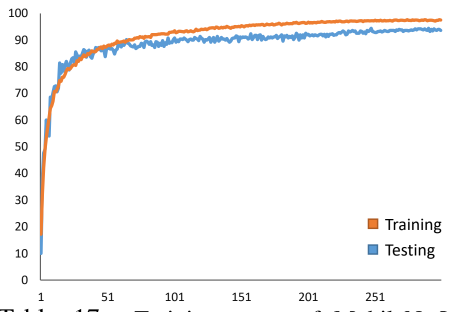


Table 17: Training curve of MobileNetV2-RNNPool on ImageNet-10.

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.

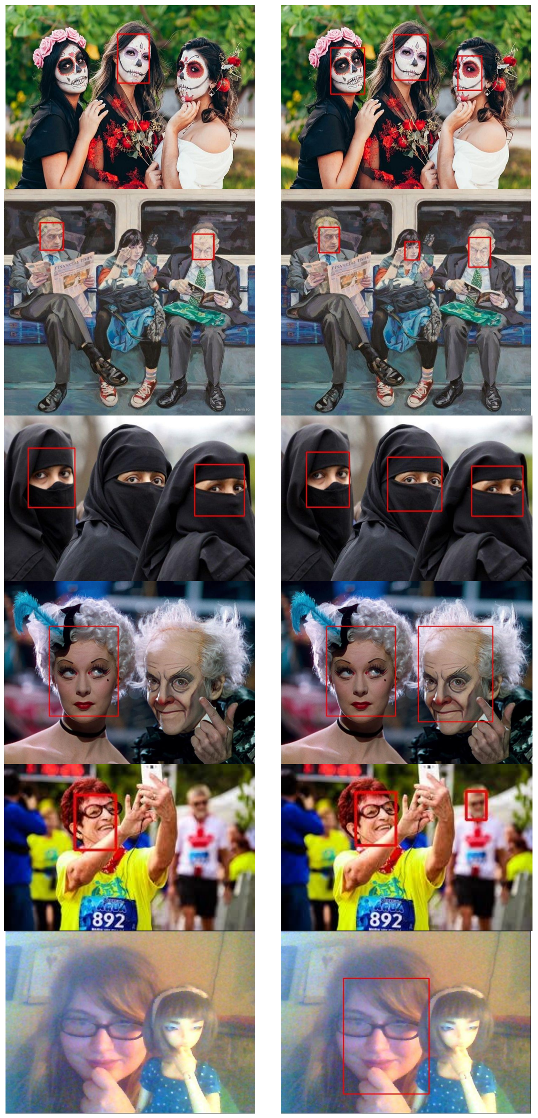


Figure 6: Comparison of performance on test images with Eagle-Eye and RNNPool-Face-Quant. The confidence threshold is set to 0.6 for both models. EagleEye misses faces when there is makeup, occlusion, blurriness and in grainy pictures, while our method detects them. However, in the case of some hard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

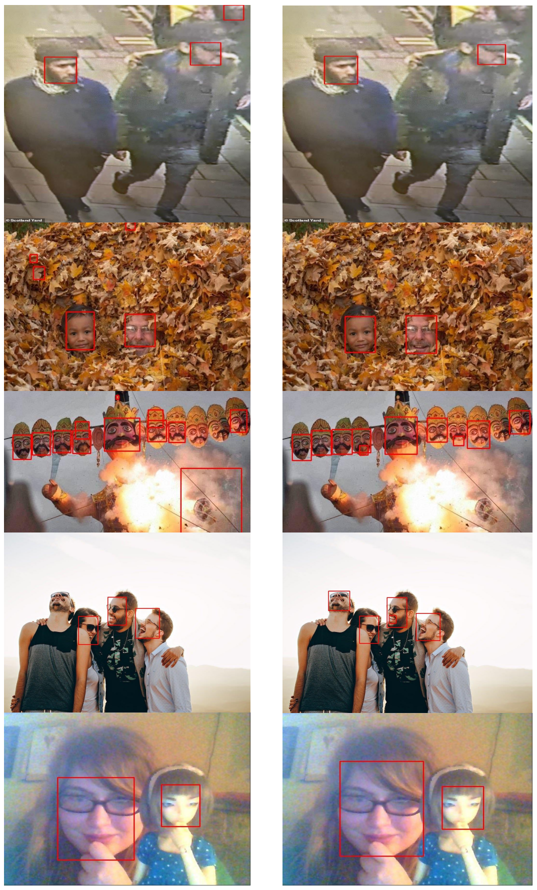


Figure 7: Comparison of performance on test images with EXTD_32 and RNNPool-Face-C. The confidence threshold is set to 0.6 for both models. The EXTD model has more false positives and misses more faces. In the first image, EXTD makes a faulty prediction at the top right. In the second image, EXTD mistakes 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.