Pick-or-Mix: Dynamic Channel Sampling for ConvNets
Supplement

Ashish Kumar[†], Daneul Kim[‡], Jaesik Park[‡], Laxmidhar Behera[†]
[†] Indian Institute of Technology Kanpur, India
[‡] Seoul National University, Republic of Korea

A. PiX Instantiation

Figure 1 shows how one can use PiX in different network architectures and for different tasks.

B. Difference with Existing Modules

Figure 2 shows visual differences with the existing modules which aims for accuracy improvement and dynamic pruning approaches.

C. Computational Complexity

We show how PiX achieves computationally efficient channel sampling. However, for better understanding, we first discuss the FLOPs of different kinds of layers.

C.1. Convolution

Consider a convolution layer having N kernels and an input feature map $X \in \mathbb{R}^{C \times H \times W}$. The size of each kernel can be given by $C \times k \times k$. FLOPs for convolution operation is determined using Fusion-Multi-Addition (FMA) instructions. Therefore, the computational demands of a convolution layer can be given as follows:

$$\#\text{FLOPs} = H \times W \times N \times C \times k \times K \tag{1}$$

C.2. BatchNorm

The BatchNorm [4] operation is performed per spatial location and can be given as $\hat{X} = (X - \mu)\frac{\gamma}{\sigma} + \beta$. It can be implemented in three FLOPs, i.e., first for computing $X - \mu$, second for γ/σ, and last as FMA with β. In general, σ is stored as σ^2, therefore, it requires to compute square-root of σ^2 to obtain σ. Overall, it takes four FLOPs to implement a BatchNorm operation per spatial location. Thus, the total number of FLOPs for a BatchNorm layer can be given as:

$$\#\text{FLOPs} = 4 \times C \times H \times W \tag{2}$$

Optionally, during inference, BN can be fused with a Conv operation where convolution is followed by BN, but we remain agnostic to such cases to account for the training phase and other architectures.

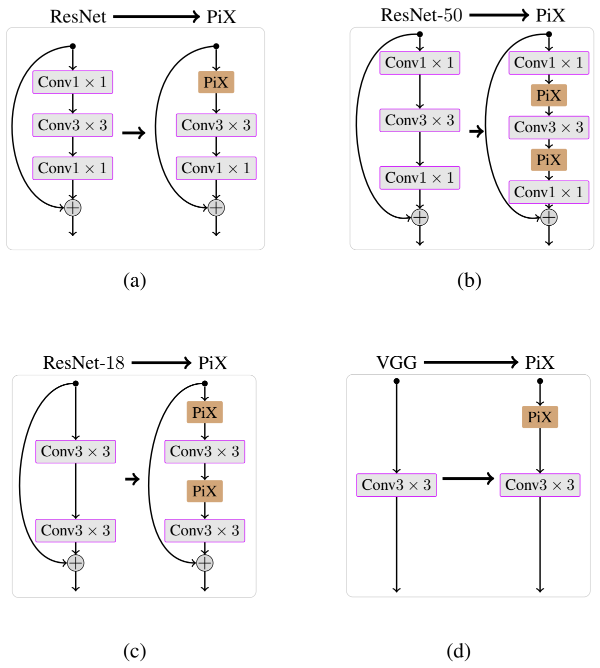

Figure 1. Embedding the proposed PiX into various standard networks for various purposes. (a) **Channel Squeezing Mode**: we replace 1×1 channel squeezing layers in ResNet [2] with PiX, where the remaining 1×1 conv layers in the original ResNet are untouched as it is intended for expanding channel dimensions. (b & c) **Network Downscaling Mode**: We insert PiX modules into ResNet and VGG [9]. We make the output channel dimension smaller than the input channel dimension by adjusting sampling factor ζ in PiX. In other words, depending on ζ, The input and output channel dimensions of 1×1 and 3×3 conv layers change accordingly. As a result, as ζ gets larger, the channel dimension of the original network reduces. (c & d) **Dynamic Channel Pruning**: These configurations are used for comparing PiX with other dynamic channel pruning approaches.

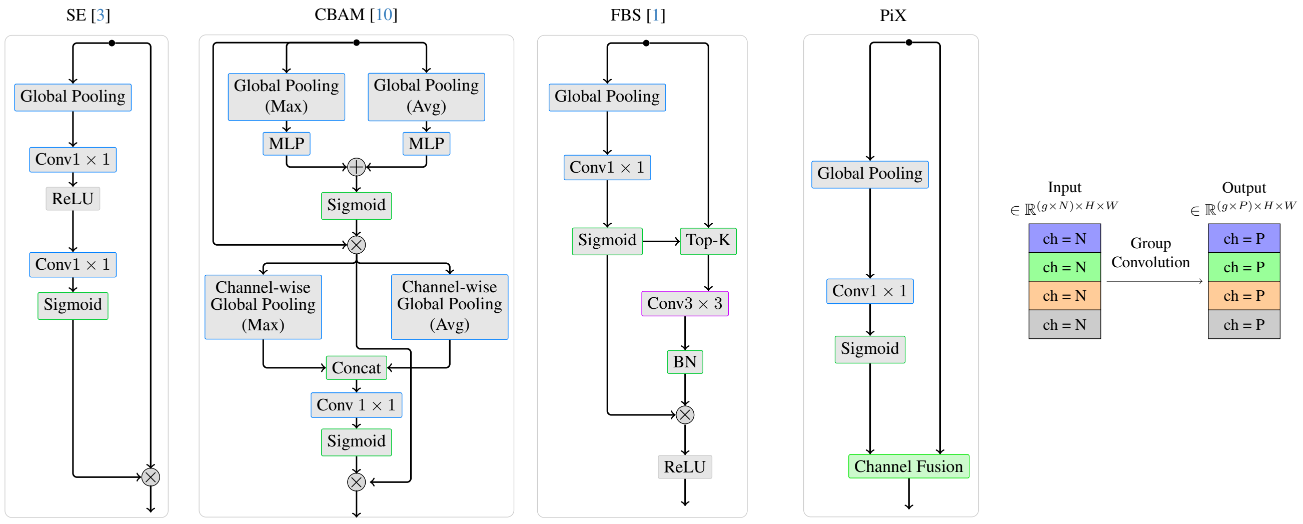

Figure 2. PiX vs existing modules: SE [3], CBAM [10], FBS [1], and Group convolution [6, 11].

C.3. ReLU

A ReLU operation is given by $Y = X$ for $X \geq 0$ and $Y = 0$ for $X < 0$. It simply requires a comparison instruction, leading to the total number of FLOPs given by:

$$\#\text{FLOPs} = C \times H \times W \tag{3}$$

C.4. Sigmoid

A Sigmoid operation is given by $Y = {}^{1}/_{(1+\exp^{-X})}$. It can be implemented in four FLOPs. Therefore, the total FLOPs for a Sigmoid layer can be given by:

$$\#\text{FLOPs} = 4 \times C \times H \times W \tag{4}$$

C.5. Global pooling

Apart from the above layers, in the PiX module, a global pooling operation is also performed. There are several ways to implement a global pooling operation. However, the most common is by using matrix multiplication routines and Fused-Multiply-Add (FMA) instructions. The whole channel of a feature map can be considered as a vector of size $H \times W$ which can be reduced to a scalar by taking its dot product with a vector whose all elements are equal to one. Hence, the total number of FLOPs for the global pooling operation can be given by:

$$\#\text{FLOPs} = C \times H \times W \tag{5}$$

C.6. Channel Sampling

Channel fusion operates on (C/ζ) subsets, each of ζ channels. For the `Max` operation, $(\zeta - 1)$ compare instructions, while for `Avg` operation, $(k - 1)$ FMA instructions are required per-location i.e. Γ_{hw}. Thus, the total number of FLOPs for channel sampling can be given by:

$$\#\text{FLOPs} = (\zeta - 1) \times (C/\zeta) \times H \times W \tag{6}$$

The computational complexity of the PiX block can be calculated based on the several equations developed above.

D. Computations & Memory Requirements

By using the above equations, we can easily compute the FLOP overhead of various modules such as SE [3], CBAM [10], or FBS [1] and demonstrated below:

D.1. SE [3]

Compute

$$\#\text{Global\_pool\_FLOPs} = C \times H \times W \tag{7}$$
$$\#\text{Conv\_Sqz\_FLOPs} = (C/16) \times C \tag{8}$$
$$\#\text{ReLU\_FLOPs} = (C/16) \tag{9}$$
$$\#\text{Conv\_Exp\_FLOPs} = C \times (C/16) \tag{10}$$
$$\#\text{Sigmoid\_FLOPs} = 4 * C \tag{11}$$
$$\#\text{Broadcast\_Multiply\_FLOPs} = C \times H \times W \tag{12}$$

$$\#\text{Total Flops} = 2CHW + 0.125C^2 + (65/16)C.$$

Memory

$$\#\text{Global\_pool\_Mem} = C \tag{13}$$
$$\#\text{Conv\_Sqz\_Mem} = C/16 \tag{14}$$
$$\#\text{Conv\_Exp\_Mem} = C \tag{15}$$
$$\#\text{Broadcast\_Multiply\_Mem} = C \times H \times W \tag{16}$$

$\#$Total Memory $= CHW + (33/16)C$.

Note: ReLU and Sigmoid are ignored in memory due to their In-place operations.

D.2. CBAM [10]

Compute

$$\#\text{Global\_Max\_pool\_FLOPs} = C \times H \times W \tag{17}$$
$$\#\text{Global\_Avg\_pool\_FLOPs} = C \times H \times W \tag{18}$$
$$\#\text{Conv\_Sqz\_FLOPs} = (C/16) \times C \tag{19}$$
$$\#\text{ReLU\_FLOPs} = (C/16) \tag{20}$$
$$\#\text{Conv\_Exp\_FLOPs} = C \times (C/16) \tag{21}$$
$$\#\text{Sigmoid\_FLOPs} = 4 * C \tag{22}$$
$$\#\text{Sum\_FLOPs} = C \tag{23}$$
$$\#\text{Broadcast\_Multiply\_FLOPs} = C \times H \times W \tag{24}$$
$$\#\text{Channel\_Max\_Pool\_FLOPs} = (C-1) \times H \times W \tag{25}$$
$$\#\text{Channels\_Avg\_Pool\_FLOPs} = (C-1) \times H \times W \tag{26}$$
$$\#\text{Concat\_FLOPs} = 2 \times H \times W \tag{27}$$
$$\#\text{Conv\_FLOPs} = 1 \times 2 \times H \times W \tag{28}$$
$$\#\text{Sigmoid\_FLOPs} = 4 \times 1 \times H \times W \tag{29}$$
$$\#\text{Broadcast\_Multiply\_FLOPs} = C \times H \times W \tag{30}$$

$\#$Total Flops $= 6CHW + 0.125C^2 + (81/16)C + 6HW$.

Memory

$$\#\text{Global\_Max\_pool\_Mem} = C \tag{31}$$
$$\#\text{Global\_Avg\_pool\_Mem} = C \tag{32}$$
$$\#\text{Conv\_Sqz\_Mem} = C/16 \tag{33}$$
$$\#\text{Conv\_Exp\_Mem} = C \tag{34}$$
$$\#\text{Sum\_Mem} = C \tag{35}$$
$$\#\text{Broadcast\_Multiply\_Mem} = C \times H \times W \tag{36}$$
$$\#\text{Channel\_Max\_Pool\_Mem} = H \times W \tag{37}$$
$$\#\text{Channels\_Avg\_Pool\_Mem} = H \times W \tag{38}$$
$$\#\text{Concat\_Mem} = 2 \times H \times W \tag{39}$$
$$\#\text{Conv\_Mem} = H \times W \tag{40}$$
$$\#\text{Broadcast\_Multiply\_Mem} = C \times H \times W \tag{41}$$

$\#$Total Memory $= 2CHW + 5HW + (65/16)C$.

D.3. FBS [1]

Compute

$$\#\text{Global\_pool\_FLOPs} = C \times H \times W \tag{42}$$
$$\#\text{Conv\_Sqz\_FLOPs} = C \times C \tag{43}$$
$$\#\text{Sigmoid\_FLOPs} = 4 \times C \tag{44}$$
$$\#\text{Top-k\_FLOPs} = \sum_{i \in [1,k]} (C - i) \tag{45}$$
$$\#\text{BatchNorm\_FLOPs} = 4 \times C \times H \times W \tag{46}$$
$$\#\text{Broadcast\_Multiply\_FLOPs} = C \times H \times W \tag{47}$$
$$\#\text{ReLU\_FLOPs} = C \times H \times W \tag{48}$$

$\#$Total Flops $= 7CHW + C^2 + 4C + \sum_{i \in [1,k]} (C - i)$.

Memory

$$\#\text{Global\_pool\_Mem} = C \tag{49}$$
$$\#\text{Conv\_Sqz\_Mem} = C \tag{50}$$
$$\#\text{Top-k\_Mem} = C \times H \times W \tag{51}$$
$$\#\text{Broadcast\_Multiply} = C \times H \times W \tag{52}$$

$\#$Total Memory $= 2CHW + 2C$.

Note: In memory, BatchNorm is ignored due to its In-place operations.

D.4. PiX

Compute

$$\#\text{Global\_pool\_FLOPs} = C \times H \times W \tag{53}$$
$$\#\text{Conv\_Sqz\_FLOPs} = (C/\zeta) \times C \tag{54}$$
$$\#\text{Sigmoid\_FLOPs} = 4 * (C/\zeta) \tag{55}$$
$$\#\text{Chanl\_Fusion\_FLOPs} = (\zeta - 1) \times (C/\zeta) \times H \times W \tag{56}$$

$\#$Total Flops $= CHW + \frac{C^2}{\zeta} + 4(C/\zeta) + ((\zeta - 1)/\zeta)CHW$.
$\#$Total Flops$(@\zeta = 1) = CHW + C^2 + 4C$.

Memory

$$\#\text{Global\_pool\_Mem} = C \tag{57}$$
$$\#\text{Conv\_Sqz\_Mem} = C/\zeta \tag{58}$$
$$\#\text{Channel Fusion Mem} = C \times H \times W \tag{59}$$

$\#$Total Memory $= CHW + ((1 + \zeta)/\zeta)C$.

From the above equations, it can be seen that PiX has the lowest FLOPs and Memory required compared to all the approaches. Values are highlighted in Table 1.

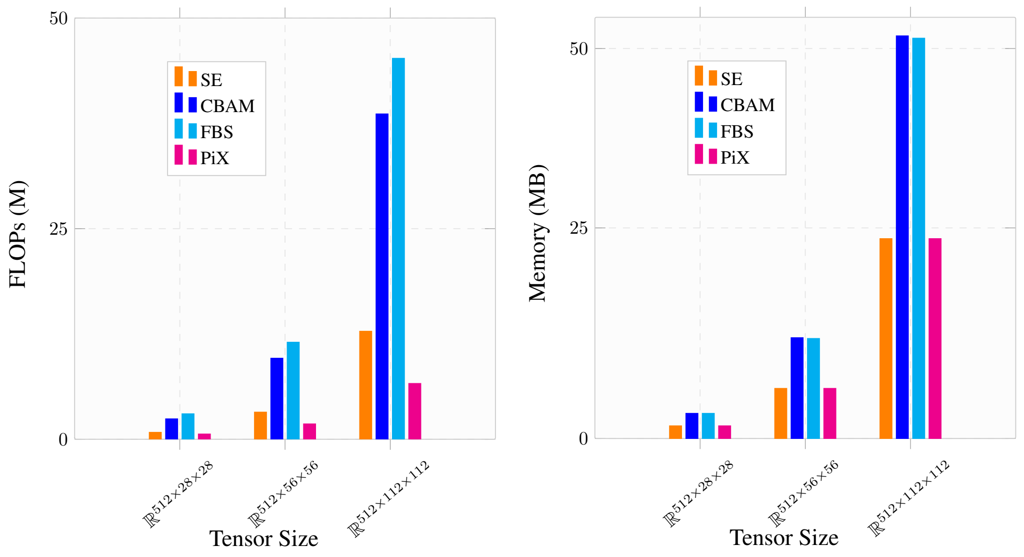

Figure 3. Flops and Memory performance of PiX in contrast to SE [3] CBAM [10], and FBS [1] per-instance of a module. In the memory plot, SE and PiX has almost same overhead but PiX lesser than SE in terms of Bytes (∼ 1,000), and same is with CBAM and FBS. For this reason plots are overlapping in the memory plot. The actual values are also highlighted in Table 1.

Table 1. This table shows FLOPs and memory usage per instance of different modules corresponding to Figure 3. These values are computed at different heights and widths of the tensor. It can be seen that PiX has the lowest FLOP overhead and also requires less memory, equivalent to SE [3] but half of CBAM [10] and FBS [1].

	@$\mathbb{R}^{512\times112\times112}$	
Method	#FLOPs (M)	#Memory (MB)
● SE [3]	12.8	25.694336
● CBAM [10]	38.6	51.639424
● FBS [1]	45.2	51.384320
○ PiX	**6.6**	**25.694208**
	@$\mathbb{R}^{512\times56\times56}$	
Method	#FLOPs (M)	#Memory (MB)
● SE [3]	3.2	6.426752
● CBAM [10]	9.6	12.916096
● FBS [1]	11.5	12.849152
○ PiX	**1.8**	**6.426624**
	@$\mathbb{R}^{512\times28\times28}$	
Method	#FLOPs (M)	#Memory (MB)
● SE [3]	.837	1.609856
● CBAM [10]	2.4	3.235264
● FBS [1]	3.0	3.215360
○ PiX	**0.6**	**1.609728**

E. Computation Reduction by PiX in Channels Squeezing i.e. $\zeta > 1$

In the baseline method, the squeeze layer operates upon $\mathbf{X} \in \mathbb{R}^{C\times H\times W}$ which requires $C/\zeta \times C \times H \times W$ FLOPs. Whereas in PiX, the global context aggregation requires $C \times H \times W$ FLOPs, cross-channel information blending requires $C/\zeta \times C$ FLOPs. and channel fusion requires $C/\zeta \times (\zeta - 1) \times H \times W$ FLOPs.

As an example, consider an input tensor $\mathbf{X} \in \mathbb{R}^{12\times5\times5}$ to a squeeze layer kernels of size 1×1. With $\zeta = 4$, the number of subsets becomes $12/\zeta = 3$. From the equations discussed, the total number of FLOPs for a squeeze layer equals 1,275.

$$\#\text{Conv\_FLOPs} = 5 \times 5 \times 3 \times 12 \times 1 \times 1 = 900 \quad (60)$$
$$\#\text{BN\_FLOPs} = 4 \times 3 \times 5 \times 5 = 300 \quad (61)$$
$$\#\text{ReLU\_FLOPs} = 3 \times 5 \times 5 = 75 \quad (62)$$

On the other hand, the FLOPs for the PiX module with $\zeta = 4$ equals only 811, as described below.

$$\#\text{Pooling\_FLOPs} = 12 \times 5 \times 5 = 300 \quad (63)$$
$$\#\text{Conv\_FLOPs} = 1 \times 1 \times 3 \times 12 \times 1 \times 1 = 36 \quad (64)$$
$$\#\text{Sigmoid\_FLOPs} = 4 \times 3 \times 1 \times 1 = 12 \quad (65)$$
$$\#\text{Sampling\_FLOPs} = 3 \times 3 \times 5 \times 5 = 225 \quad (66)$$

In the above example, the baseline squeezing method requires 1,275 FLOPs, whereas PiX requires only 523 and 748 FLOPs for PiX and w-PiX fusion strategy respectively. In a similar manner, we achieve huge gains when PiX is plugged into the existing networks, which have been discussed in the experiments section of the paper.

F. Effect of Pick-or-Mix on Memory in Channel Squeezing

Despite the computational benefits, PiX does not introduce any memory overhead. The total memory required by the

Table 2. Ablation study of ResNet-50 + PiX@$\zeta = 4$. Top-1 Accuracy on ImageNet.

	Ablation	Parameter	Top-1 Accuracy
E0	Fusion Activation	Sigmoid	76.77%
		TanH	76.39%
E1	Batch-Norm	✗	76.77%
		✓	76.44%
E2	τ	0.0	76.58%
		0.5	76.77%
		1.0	76.54%
E3	Operator	`Min`	74.68%
		`Max`	76.57%
		`Avg`	76.58%
		`Max+Avg`	76.77%

baseline squeeze operation with $\zeta = 4$ can be given by: #M $= C/4 \times H \times W$. On the other hand, the memory required for PiX is given by: #M $= C + C/4 + C/4 \times H \times W$. We can see that there is a negligible increment in the memory footprint, i.e., from $0.75 \times C \times H \times W$ to $0.75 \times C \times H \times W + 1.25C$. For FP32 precision, the raw memory footprint will be $4 \times M$.

G. Ablation Study

We empirically validate Pick-or-Mix design practices using the most pertinent ablations possible. ResNet-50 is adopted as the baseline for this purpose, and channel squeezing mode. To begin with, we first analyze the effect of changing the activation function in the cross-channel information blending stage and then examine the effect of placing a BatchNorm prior to the sigmoidal activation. Further, we verify the behavior of proposed channel fusion strategies and also the effect of varying fusion threshold τ.

E0: Fusion Activation. The channel fusion stage utilizes the sampling probability p. Given that the value of p lies in the interval $[0, 1]$, we wish to examine the behavior of PiX if this range is achieved via a different activation function. For this purpose, we select `TanH` function which natively squeezes the input into a range $[-1, 1]$. Therefore, we rewrite the mathematical expression to $0.5 \times (1 + \texttt{TanH})$ in order to place the output of TanH into the desired range of $[0, 1]$. We replace the sigmoidal activation with the above expression and retrain the network. From Table 2, it can be seen that sigmoidal activation outperforms the `TanH` activation for the case of PiX.

E1: BatchNorm in Global Context Aggregation. Out of curiosity, we also analyze the behavior of PiX module by placing a BatchNorm [4] after the sampling probability predictor because the squeeze layer in the baseline method is also followed by a BatchNorm layer. We observe that BatchNorm negatively impacts performance.

E2: Effect of Fusion Threshold (τ). The hyperparameter τ is evaluated against three values $\in \{0.0, 0.5, 1.0\}$. In accordance with Eq. 2 of the main manuscript, $\tau = 0$ corresponds to `Max` operator, $\tau = 1.0$ corresponds to `Avg` operator regardless of the value of p. Whereas $\tau = 0.5$ offers equal opportunity to the `Max` and `Avg` fusion operators which are adaptively taken care of by the value of p. We present an ablation over the aforementioned three values of τ. From Table 2, we observe that $\tau = 0.5$ results in best performance, which is the case when the network has the flexibility to choose from both reduction operators adaptively. Hence, in the experiments, we use $\tau = 0.5$ for threshold-based fusion.

E4: Effect of Operator Type. We also experiment for operator `Min` other than `Max` and `Avg`. We found out that `Min` performs severely worse. This justifies our choice of operators and is in line with the performance achieved by using the pooling operation when they are used spatially.

H. Role of Fusion Probability

We analyze the sampling probabilities across all classes in the ImageNet validation set for ResNet-50 + PiX @$\zeta = 2$ for the last block of each stage (Figure 4).

It can be seen that the importance of probability is significant since distribution for the fusion operator selection is variable, i.e., while training, the network does not bias towards only one type of fusion operator, indicating that both of the fusion operators are crucial. In the deeper layers (stage-5), the variance starts increasing, indicating deeper layers are class-specific and need different activation distributions. This is in line with [3]. Moreover, we notice that, unlike [3], none of the layers in the stage-5 show saturation. This is also an indication that PiX naturally pushes a convolution layer to learn more complex representation.

I. GradCAM Visualization

The performance of PiX, especially in the channel squeezing mode, inspires us to analyze how PiX attends the spatial regions relative to the baseline. It explains qualitatively the improved performance of PiX despite the reduction in FLOPs. We use GradCAM [8] for this purpose.

Figure 5 shows the analysis for ResNet and VGG. Noticeably, PiX shows improvement in the attended regions of a target class relative to the baseline (R-I2, V-I4). Also, in images with multiple instances, PiX focuses on each instance strongly (R-I4, V-I2), indicating that PiX enhances network's generalization by learning to emphasize class-specific parts.

J. GPU Deployment for Pick-or-Mix

The implementation of PiX is quite straightforward and fully parallelizable. The sampling probability and output feature

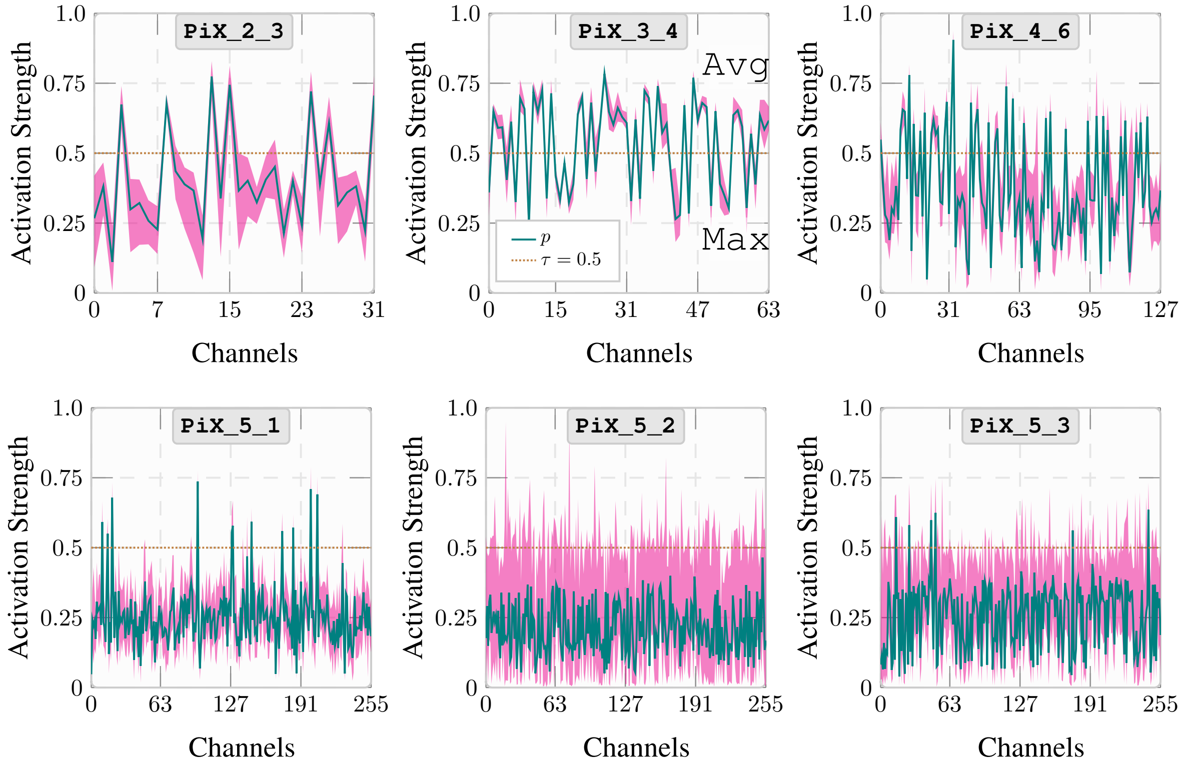

Figure 4. Sampling probability at different stages of ResNet-50 + PiX. Stage named as: `PiX_STAGE_ID_BLOCK_ID` [2].

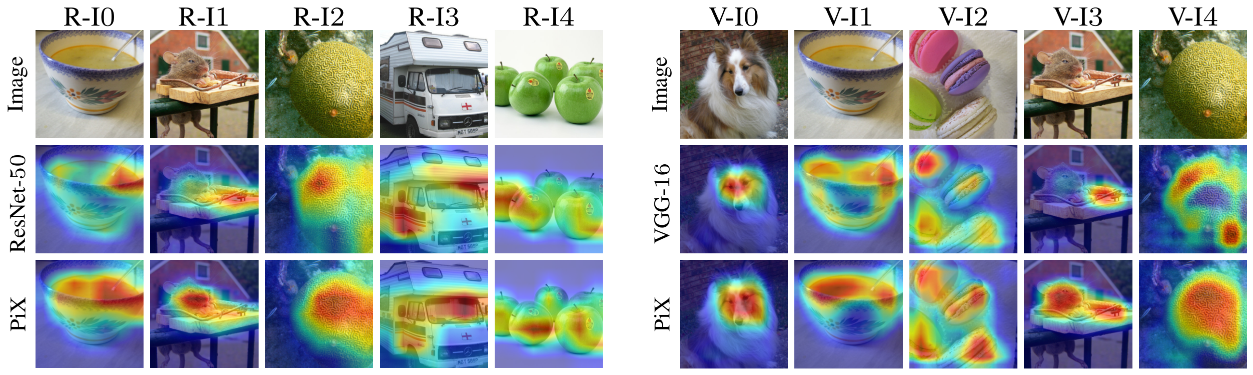

Figure 5. GradCAM for ResNet-50 + PiX, VGG-16 + PiX. solid red shows more confidence for a pixel to belong to a class.

map computations are parallelizable because they are pointwise operations.

PiX can be implemented directly with the fundamental operators of Pytorch [7]. However, since we perform operations over each subset and each location independently, therefore, PiX requires merely $10 - 15$ lines of NVIDIA's CUDA kernel code or any other parallelization paradigm.

K. Codes and Implementation

The code and the pre-trained models are open-sourced in PyTorch [7]. See below for Python and CUDA snippets.

L. Training Specifications.

The training procedure is kept standard to ensure reproducibility. We use a batch size of 256, which is split across 8 GPUs. We use a RandomResized crop [7] of 224×224 pixels, along with a horizontal flip. We use `SGD` with `Nesterov` momentum of 0.9, `base_lr`=0.1 with CosineAnnealing [5] rate scheduler and a weight decay of 0.0001. Unless otherwise stated, all models are trained from scratch for 120 epochs following [2].

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import pix_layer_cuda
import math

# gradients in the backward are received in the order of tensor as they were output in forward function
class PiXOperator(torch.autograd.Function):
    @staticmethod
    def forward(ctx, zeta: int, tau: float,  input: torch.Tensor, fusion_prob: torch.Tensor):
        outputs = pix_layer_cuda.forward(zeta, tau, input, fusion_prob)
        ctx.save_for_backward(input, fusion_prob)
        ctx.zeta = zeta
        ctx.tau = tau
        return outputs[0]

    @staticmethod
    def backward(ctx, out_grad):
        input, fusion_prob = ctx.saved_tensors
        zeta = ctx.zeta
        tau = ctx.tau
        input_grad, fusion_prob_grad = pix_layer_cuda.backward(zeta, tau, input, fusion_prob, out_grad)
        return None, None, input_grad, fusion_prob_grad


class PiXOperatorLayer(torch.nn.Module):
    def __init__(self, zeta, tau = 0.5):
        super(PiXOperatorLayer, self).__init__()
        self.zeta = int(zeta)
        self.tau = tau

    def forward(self, input, fusion_prob):
        return PiXOperator.apply(self.zeta, self.tau, input, fusion_prob)

class PiXLayer(torch.nn.Module):
    def __init__(self, n_ip, zeta, tau=0.5):
        super(PiXLayer, self).__init__()

        n_op = math.ceil(float(n_ip) / zeta)
        self.conv1x1 = torch.nn.Conv2d(n_ip, n_op, 1)

        self.pix = PiXLayer(zeta, tau)
        self.global_pool = torch.nn.AdaptiveAvgPool2d((1, 1))
        self.sigmoid_sqz = torch.nn.Sigmoid()

    def forward(self, x):
        global_pool = self.global_pool(x)
        conv_g_pool = self.conv1x1(global_pool)
        sos_likelihood = self.sigmoid_sqz(conv_g_pool)
        x = self.pix.forward(x, sos_likelihood)
        return x



#### USAGE

n_ip = 24
zeta = 4
pix = PiXLayer(n_ip, zeta)
X = torch.ones([1, n_ip, 4, 4])
Y = pix(X)
print(X)
print(Y)
```

```
#include <torch/extension.h>
#include <ATen/cuda/CUDAContext.h>
#include <ATen/ops/matmul.h>

#include <vector>
#include<iostream>

const int MAX_THREADS_PER_BLOCK = 512;

template <typename Dtype>
__global__ void PiX_Forward_cuda_kernel(const int n_threads,
                                   int out_channels, int in_channels,
                                   int height, int width,
                                   int k_size,
                                   float TAU,
                                   const Dtype* __restrict__ bottom_data,
                                   const Dtype* __restrict__ prob_data,
                                   Dtype* __restrict__ top_data)
{
   int thread_idx = blockIdx.x * blockDim.x + threadIdx.x;
   if(thread_idx < n_threads)
   {
        unsigned int n = thread_idx / (out_channels*height*width);
        unsigned int c = (thread_idx / (height*width)) % out_channels;
        unsigned int h = (thread_idx / (width)) % height;
        unsigned int w = thread_idx % width;

        Dtype prob = prob_data[n*out_channels+c];

        int c_start = c * k_size;
        int c_end   = c_start + k_size;
        if(c_end > in_channels)
        c_end = in_channels;

        if(prob < TAU)
        {
            Dtype max_val = -FLT_MAX;

            for(int ch = c_start; ch < c_end; ch++)
            {
                unsigned long int bottom_index = ((n*in_channels+ch)*height+h)*width+w;

                Dtype bottom_val = bottom_data[bottom_index];
                if(max_val < bottom_val)
                {
                    max_val = bottom_val;
                }
            }

            top_data[thread_idx] = prob * max_val;
        }
        else
        {
            Dtype avg_val = 0;

            for(int ch = c_start; ch < c_end; ch++)
            {
                unsigned long int bottom_index = ((n*in_channels+ch)*height+h)*width+w;

                avg_val += bottom_data[bottom_index];
            }

            top_data[thread_idx] =  (prob) * avg_val / k_size;

        }
    }
}


template <typename Dtype>
__global__ void PiX_Backward_cuda_kernel(const int n_threads,
                                    int out_channels, int in_channels,
                                    int height, int width,
                                    int k_size,
                                    float TAU,
                                    const Dtype* __restrict__ bottom_data,
                                    const Dtype* __restrict__ top_diff,
                                    const Dtype* __restrict__ prob_data,
                                    Dtype* __restrict__ prob_diff,
                                        Dtype* __restrict__ bottom_diff)
{
   int thread_idx = blockIdx.x * blockDim.x + threadIdx.x;
   if(thread_idx < n_threads)
   {

        unsigned int n = thread_idx / (out_channels*height*width);
        unsigned int c = (thread_idx / (height*width)) % out_channels;
        unsigned int h = (thread_idx / (width)) % height;
        unsigned int w = thread_idx % width;

        Dtype prob = prob_data[n*out_channels+c];

        int c_start = c * k_size;
        int c_end   = c_start + k_size;
        if(c_end > in_channels)
        c_end = in_channels;


        if(prob < TAU)
        {
            Dtype max_val = -FLT_MAX;
            int max_idx = c_start;

            for(int ch = c_start; ch < c_end; ch++)
            {
                unsigned long int bottom_index = ((n*in_channels+ch)*height+h)*width+w;

                Dtype bottom_val = bottom_data[bottom_index];
                if(max_val < bottom_val)
                {
                    max_val = bottom_val;
                    max_idx = ch;
                }
            }

            Dtype top_diff_val = top_diff[thread_idx];

            for(int ch = c_start; ch < c_end; ch++)
            {
                if(ch == max_idx)
                    bottom_diff[((n*in_channels+ch)*height+h)*width+w] = prob * top_diff_val;
                else
                    bottom_diff[((n*in_channels+ch)*height+h)*width+w] = 0;
            }

            prob_diff[((n*out_channels+c)*height+h)*width+w] = max_val * top_diff_val;
        }
        else
        {
            Dtype top_diff_val = top_diff[((n*out_channels+c)*height+h)*width+w];
            Dtype avg_val = 0;

            for(int ch = c_start; ch < c_end; ch++)
            {
                bottom_diff[((n*in_channels+ch)*height+h)*width+w] = (prob) * top_diff_val / k_size;
                avg_val +=  bottom_data[((n*in_channels+ch)*height+h)*width+w];
            }

            avg_val /= k_size;

            prob_diff[((n*out_channels+c)*height+h)*width+w] =   top_diff_val * avg_val;

        }
    }
}




std::vector<torch::Tensor> PiX_cuda_forward(
    const int k_size,
    float TAU,
    torch::Tensor input,
    torch::Tensor sampling_prob)
{

     int n = input.size(0);
```

```
      int in_c = input.size(1);
      int h = input.size(2);
      int w = input.size(3);

      int out_c = ceil((float)in_c / k_size);

 torch::Tensor output = torch::zeros({n, out_c, h , w}, input.options());

  const int n_threads = n * out_c * h * w;

  const dim3 blocks((n_threads - 1) / MAX_THREADS_PER_BLOCK + 1, 1, 1);

  c10::cuda::CUDAStream stream = c10::cuda::getCurrentCUDAStream();

  AT_DISPATCH_FLOATING_TYPES(output.type(), "PiX_cuda_forward", ([&] {
    PiX_Forward_cuda_kernel<scalar_t><<<blocks, MAX_THREADS_PER_BLOCK,0,stream>>>(
        n_threads,
        out_c, in_c,
        h, w,
        k_size,
        TAU,
        (scalar_t*)input.data_ptr(),
        (scalar_t*)sampling_prob.data_ptr(),
        (scalar_t*)output.data_ptr());
  }));


  return {output};
}

std::vector<torch::Tensor> PiX_cuda_backward(
    const int k_size,
    float TAU,
    torch::Tensor input,
    torch::Tensor sampling_prob,
    torch::Tensor output_grad)
{
       int n = input.size(0);
      int in_c = input.size(1);
      int h = input.size(2);
      int w = input.size(3);

      int out_c = ceil((float)in_c / k_size);

  torch::Tensor input_grad = torch::zeros_like(input);
  torch::Tensor sampling_prob_grad = torch::zeros_like(output_grad);

  const int n_threads = n * out_c * h * w;

  const dim3 blocks((n_threads - 1) / MAX_THREADS_PER_BLOCK + 1, 1, 1);

  c10::cuda::CUDAStream stream = c10::cuda::getCurrentCUDAStream();

  AT_DISPATCH_FLOATING_TYPES(output_grad.type(), "PiX_cuda_backward", ([&] {
    PiX_Backward_cuda_kernel<scalar_t><<<blocks, MAX_THREADS_PER_BLOCK,0,stream>>>(
        n_threads,
        out_c, in_c,
        h, w,
        k_size,
        TAU,
        (scalar_t*)input.data_ptr(),
        (scalar_t*)output_grad.data_ptr(),
        (scalar_t*)sampling_prob.data_ptr(),
        (scalar_t*)sampling_prob_grad.data_ptr(),
        (scalar_t*)input_grad.data_ptr()
        );
  }));

  torch::Tensor ones = torch::ones({h*w, 1}, input.options());
  sampling_prob_grad = sampling_prob_grad.reshape({n*out_c, h*w});

  torch::Tensor sampling_prob_grad_reduced = at::matmul(sampling_prob_grad, ones);
  sampling_prob_grad_reduced = sampling_prob_grad_reduced.reshape({n,out_c, 1,1});


  return {input_grad, sampling_prob_grad_reduced};
}
```

References

[1] Xitong Gao, Yiren Zhao, Łukasz Dudziak, Robert Mullins, and Cheng-zhong Xu. Dynamic channel pruning: Feature boosting and suppression. *arXiv preprint arXiv:1810.05331*, 2018. 2, 3, 4

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 1, 6

[3] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018. 2, 4, 5

[4] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015. 1, 5

[5] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016. 6

[6] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*, pages 116–131, 2018. 2

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019. 6

[8] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017. 5

[9] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. 1

[10] Sanghyun Woo, Jongchan Park, Joon-Young Lee, and In So Kweon. Cbam: Convolutional block attention module. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018. 2, 3, 4

[11] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018. 2