

# Supplemental Material to “MSViT: Dynamic Mixed-Scale Tokenization for Vision Transformers”

## A. Additional qualitative results on ImageNet

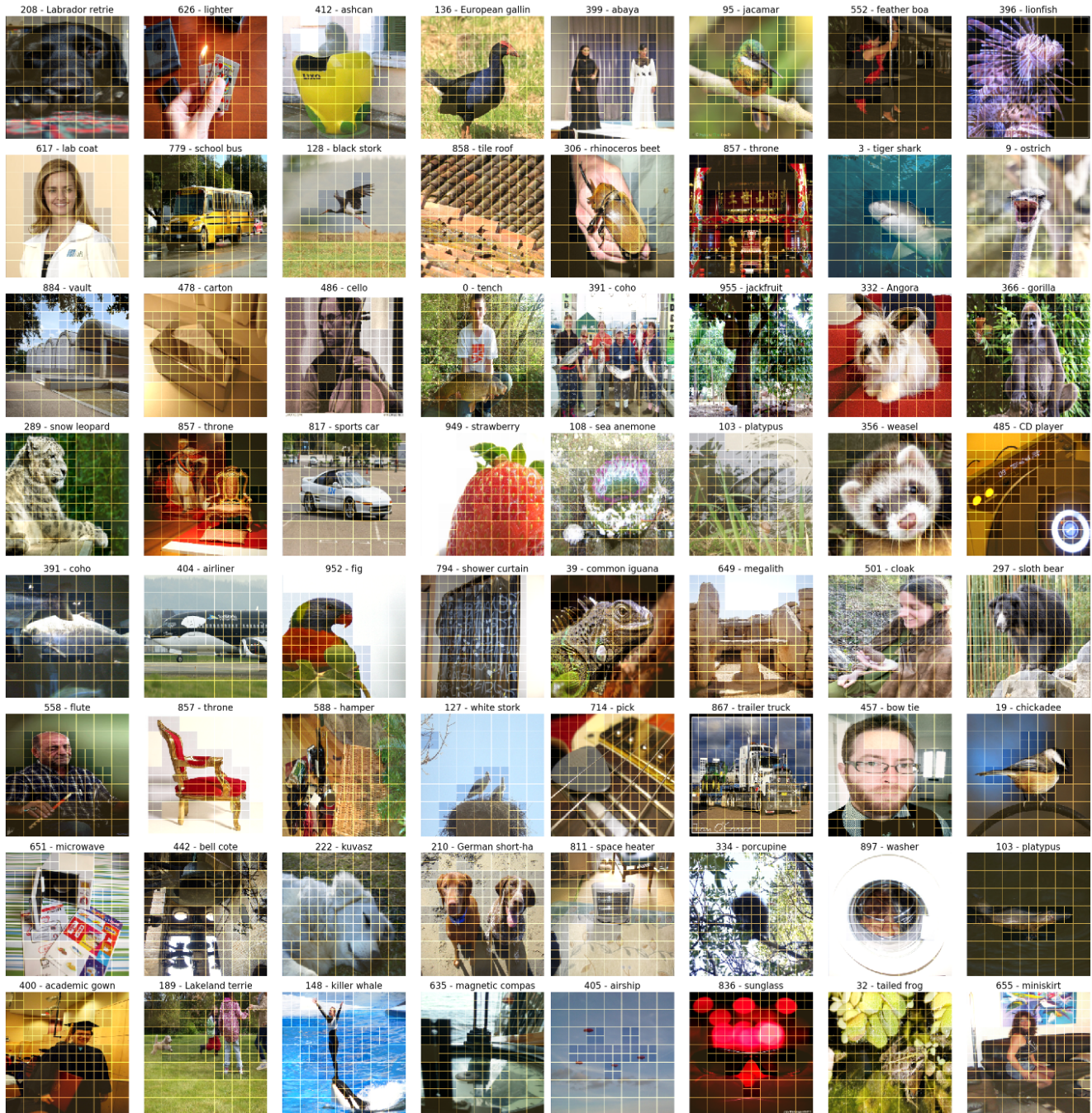


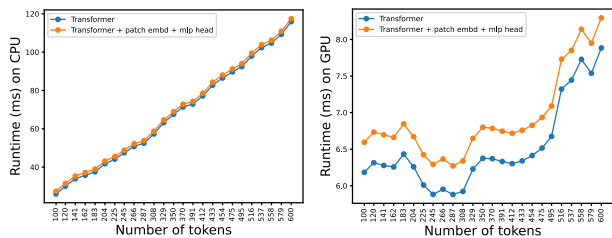


Figure 9. Non-curated qualitative examples of scale selection masks output by the gating module of MSViT-S/{16, 32}. The model was trained on 224px ImageNet images to choose between the coarse (32px, ) and the fine (16px, ) token scale. *Best seen zoomed.*

## B. Additional results on Image classification

In this section we report additional results on the ImageNet classification benchmark. First, in Figure 10, we plot the average simulated runtimes of standard ViT-S with different number of tokens. While the exact trend depends on the device due to dedicated optimizations, we verify that reducing the number of tokens leads to concrete runtime improvements.



(a) CPU (b) GPU (RTX 2080 Ti)

Figure 10. Average runtime in milliseconds of ViT-S for different number of input tokens on two different devices, simulated and averaged on 1000 random samples. The blue line is the cost of the transformer only, while the orange line additionally includes the cost of the patch embedding and MLP classifier.

Then, in Table 3, we show an extended version of Table 1, with additional results for (i) MS-ViT-L, (ii) more computational budgets and (iii) latency results averaged across the images in the ImageNet validation set.

Finally, in Figure 11, we report the results of the full hyperparameter sweep on MSViT-S for different input image sizes: As expected, both the gate sparsity target  $g^*$  and the gate loss weight  $\lambda$  can be increased to obtain sparser gates. In addition, we observe that all MSViT points lie on the same Pareto front which suggests the relative performance of MSViT is robust to these hyperparameter choices (gate loss weight, sparsity target and input image size).

## C. Hyperparameters

### C.1. ViT backbone

For ViT experiments, we finetune ImageNet-21k pre-trained checkpoints to ImageNet. We use the same finetuning setup as the one from the official ViT repository [37], except we train for 20 epochs instead of 8:

```
batch-size: 512
num-gradacc-steps: 1
data-augmentation: crop+fliplr
num-epochs: 20
optimizer: "SGD"
lr: 0.03
momentum: 0.9
gradient-clipping: 1.0
weight-decay: 0.0
```

DeiT-Small backbone	Avg # tokens	GMACs (avg)	CPU time (ms)	GPU time (ms)	accuracy	
					top-1	top-5
DeiT-S/16 in=160	100	2.27	18.70	6.06	75.86	92.84
MSDeiT-S/16,32 in=224	94	2.20	18.01	6.00	75.90	92.68
MSDeiT-S/16,32 in=224	97	2.27	18.22	6.02	76.99	93.38
DeiT-S/16 in=192	144	3.32	24.04	6.26	77.79	93.96
MSDeiT-S/16,32 in=224	116	2.72	21.18	6.28	77.79	93.99
MSDeiT-S/16,32 in=224	142	3.32	24.24	6.20	78.76	94.32
DeiT-S/16 in=224	196	4.60	31.65	6.07	79.85	94.57
MSDeiT-S/16,32 in=224	173	4.08	27.70	6.19	79.38	94.38

ViT-Tiny backbone	Avg # tokens	GMACs (avg)	CPU time (ms)	GPU time (ms)	accuracy	
					top-1	top-5
ViT-Ti/16 in=160	100	0.60	9.24	5.97	71.63	90.68
MSViT-Ti/16,32 in=224	95	0.60	8.99	5.98	72.57	91.32
ViT-Ti/16 in=192	144	0.89	11.56	6.03	74.24	92.22
MSViT-Ti/16,32 in=224	124	0.78	11.04	6.04	74.27	92.22
MSViT-Ti/16,32 in=224	138	0.88	11.49	6.00	74.93	92.54
ViT-Ti/16 in=224	196	1.25	13.26	5.98	76.00	93.26
MSViT-Ti/16,32 in=224	154	0.98	11.89	5.88	75.51	92.98

ViT-Small backbone	Avg # tokens	GMACs (avg)	CPU time (ms)	GPU time (ms)	accuracy	
					top-1	top-5
ViT-S/16 in=128	64	1.44	15.35	5.94	75.48	93.08
MSViT-S/16,32 in=224	75	1.76	16.33	5.95	77.16	94.14
ViT-S/16 in=160	100	2.27	18.60	6.06	78.88	94.95
MSViT-S/16,32 in=224	91	2.13	17.64	5.97	78.88	95.02
MSViT-S/16,32 in=224	98	2.30	18.60	6.04	79.51	95.33
ViT-S/16 in=192	144	3.32	24.11	6.18	80.75	95.86
MSViT-S/16,32 in=224	120	2.82	21.71	6.22	80.74	95.92
MSViT-S/16,32 in=224	138	3.23	23.68	6.19	81.47	96.14
ViT-S/16 in=224	196	4.60	31.46	6.08	82.02	96.45
MSViT-S/16,32 in=224	187	4.43	29.30	6.25	82.02	96.44
ViT-S/16 in=288	324	7.97	53.79	6.18	83.34	96.93
MSViT-S/16,32 in=384	314	7.92	52.67	6.02	83.56	97.10
MSViT-S/16,32 in=384	286	7.16	47.53	6.09	83.34	96.99
ViT-S/16 in=320	400	10.11	68.20	6.25	83.85	97.10
MSViT-S/16,32 in=384	359	9.19	60.16	6.18	83.84	97.20
MSViT-S/16,32 in=384	382	9.80	66.12	6.21	83.93	97.18
ViT-S/16 in=384	576	15.49	104.58	6.26	84.20	97.32
MSViT-S/16,32 in=384	428	11.14	76.76	6.16	84.14	97.31

ViT-Large backbone	Avg # tokens	GMACs (avg)	CPU time (ms)	GPU time (ms)	accuracy	
					top-1	top-5
ViT-L/16 in=160	100	31.08	185.44	12.74	81.70	96.14
MSViT-L/16,32 in=160	89	27.48	172.29	12.37	81.73	96.13
MSViT-L/16,32 in=192	84	25.93	169.63	12.46	81.67	96.14
ViT-L/16 in=192	144	44.9	233.27	14.49	82.91	96.61
MSViT-L/16,32 in=192	111	34.5	195.24	12.38	82.46	96.45

Table 3. Extended results for Table 1 with additional configurations, and average latency per image on CPU and GPU (RTX 2080 Ti). Both the MACs and latencies are estimated with the deep-speed library [38].

```
num-warmup-epochs: 0.50
lr-scheduler: cosine
```

### C.2. DeiT backbone

For DeiT, we also follow standard available finetuning pipelines e.g. from [36, 22, 33]. In particular, the most notable differences with the ViT finetuning pipeline are:

- The data loader uses a different normalization and bicubic resizing
- We use the AdamW optimizer with  $lr = 2e-5$  (after sweeping over the range  $lr \in \{5e-4, 1e-4, 2e-5\}$ )

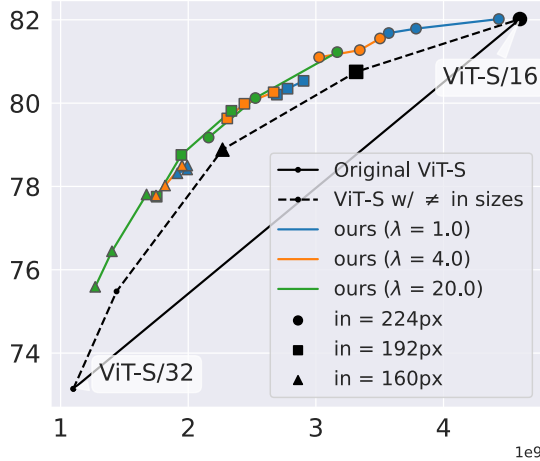


Figure 11. Full hyperparameter sweep for MSViT-S/16 experiments (top-1 accuracy versus MACs). Each line corresponds to a configuration of gate loss weight  $\lambda$  and input image size. Each point on a line corresponds to a different gate sparsity target  $g^* \in \{0.25, 0.5, 0.75\}$

- additional small optimization choices: no gradient clipping, small weight decay and label smoothing with a weight of 0.1

### C.3. Gate Hyperparameters

For training the gate, we use the same optimizer and learning rate as the model features. The only difference is that we use a longer warmup period to account for the fact that the gate is trained from scratch. For GBaS, we observe that the temperature in the Relaxed Bernoulli and the variance of the hyperprior, as long as they do not take very extreme values, do not strongly impact the final learned gate, but rather the training speed. Therefore, we fix their values in all experiments and instead sweep over the gate loss weight which also directly impacts the gate’s training speed.

`num_gate_warmup_epochs`: 6  
`relaxed_bernoulli_temperature`: 0.3  
`hyperprior_variance`: 0.1

Finally as mentioned in experiments, we sweep over the gate target  $g^* \in \{0.5, 0.25, 0.1\}$  and loss weight  $\lambda \in \{1, 4, 20\}$  to obtain models at various compute budgets.

### D. Additional segmentation results

In this section, we report additional results for the segmentation experiments. First, in Figure 12, we visualize some masks output by a ViT-S/16 finetuned on ImageNet when directly applied on 512x512 ADE20K [60] images, without extra finetuning on ADE20k. As we can see, the mixed-scale selection patterns transfer well from ImageNet to ADE20K.

Finally, we report extended results in Table 4 (same results as Figure 4 (a) in the main text but with additional latency results) and visualize some additional qualitative outputs in Figure 13.



Figure 12. Direct transfer of a gate trained on ViT-S/16 224px images for ImageNet to ADE20k for 512px images

Backbone	$g^*$	# tokens avg	MACs x 1e10	CPU time ms	GPU time ms	mIoU single-scale
Seg-T/16 (512px)	-	1024	1.04	113.68	26.5	38.1
	0.5	655	0.56	86.12	25.6	37.9
	0.25	565	0.46	75.96	25.0	37.3
MSSeg-T/16	0.1	525	0.42	69.13	24.3	36.8
Seg-S/16 (512px)	-	1024	3.17	252.09	30.7	45.3
	0.5	684	1.92	184.81	29.6	44.9
	0.25	586	1.59	153.12	29.0	44.1
MSSeg-S/16	0.1	552	1.48	144.02	28.5	43.3

Table 4. Segmentation results from Figure 4 (a) in the main text with extended timing results on (i) CPU and (ii) GPU (Tesla V100-SXM2-32GB), both reported in milliseconds

## E. Mixed-scale tokens for non-standard ViTs

In Section 4.2, we combine a pretrained mixed-scale gate with different ViT backbones. In this section, we describe how we implement these changes in more details.

### E.1. Segmenter

The **Segmenter** architecture [18] is composed of a standard ViT backbone, followed by a small decoder (either linear, or a small transformer head) to generate the segmentation map outputs. We first replace the ViT backbone by MSViT. We then simply need to recover the original spatial

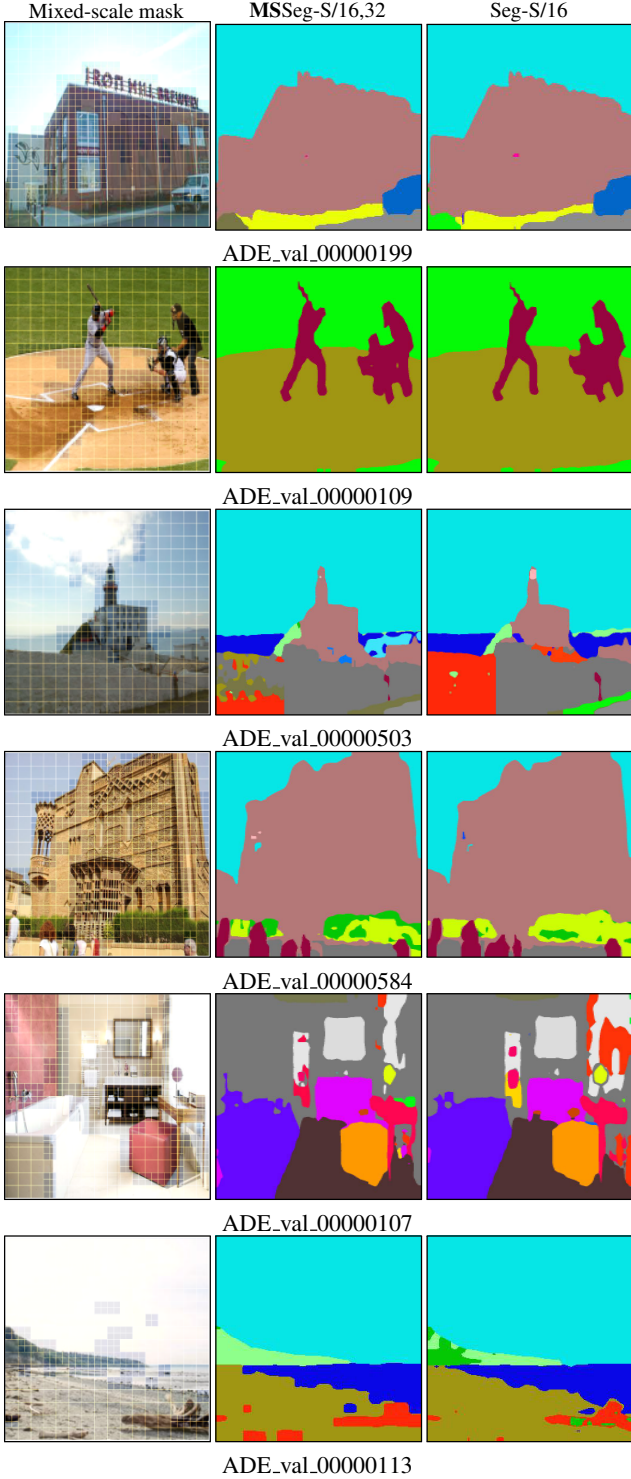


Figure 13. Non-curated qualitative results on the segmentation experiments. We display the mixed-scale mask output by the gate (left), the final segmentation map output by our MSSeg-S/16,32 trained with target  $g^* = 0.25$  (middle) and the segmentation map output by the corresponding backbone baseline Seg-S/16

resolution from the stream of mixed-scale tokens at the end of the backbone: More specifically, once the transformer backbone has been executed, we replicate every coarse token 4 times, to compensate for the fine token it replaces. We then feed this sequence of tokens to the decoder, without making any changes to its original architecture.

## E.2. Token pruning

Most SotA token pruning methods builds off the DeiT architecture, and implement some form of token binary masking, similar to how we use masked attention (Eq. 15). Thus adding mixed-scale tokenization to these models is straightforward: For instance, in **DyViT**, we simply use the binary mask output by the mixed-scale gate as the initial "pruning policy" of the model (instead of the default initialization which a mask of all ones). In **EViT**, the tokens are sorted by decreasing class-attention and a fixed ratio of the lower tokens is pruned in certain layers. We simply apply the same sort-and-prune operation to the mixed-scale mask as the one applied to the tokens and propagate them to the next layer.

## E.3. Hierarchical Transformers

Unlike ViT, hierarchical transformers such as Swin integrate multiple scale. We denote by  $s_\ell$  the scale of the  $\ell$ -th block in Swin; where each block is a sequence of transformer layers, and in practice  $s_\ell = 4 \times 2^{\ell-1}$ . The transition from a block to the next is done with a Patch Merging operation: Groups of 4 neighboring tokens are concatenated together then linearly embedded to form a unique token. As a result, as we transition through block, the number of tokens decreases (i.e., the patch scale increases) and the number of channels increases, similar to the usual CNN architecture design. In addition, Swin implements local attention is computed across windows of  $w \times w$  tokens ( $w = 7$ ).

Given a pretrained mixed-scale gate with coarse scale  $S_c$ , we first run it on the input image: This yields a binary decision for each  $S_c \times S_c$  coarse patch in the image. We use this binary mask to guide the flow of tokens through the Swin transformer: Every input token that falls in a fine scale region follows the standard Swin paradigm. For the rest of the tokens (coarse scale regions), we feed them to a simple linear embedding, and reintegrate them to the flow of fine tokens in the  $\ell$ -th block such that  $s_\ell = S_c$ .

In summary, the mixed-scale gate decides whether a token should be processed at a fine-grained level (early layers of the Swin transformer with small receptive field). The gain in computational cost comes from (i) coarse tokens skipping FFNs in the early layers, and (ii) due to the absence of coarse tokens in the early layers some local attention windows are empty, hence can be entirely skipped.

Finally, there is an interesting interaction between the base patch scale  $s_1$ , the attention window scale  $w = 7$  and the coarse scale of the gate ( $S_c = s_\ell$ ), as they all impact

the scale of the tokens. In our experiments, we consider varying the parameter  $\ell$  and show that it directly impacts the MACs-accuracy trade-off.

## F. Training dynamics of adaptive trimming

In [Section 2.3](#) we introduce the adaptive trimming strategy (**AT**) for reducing training overhead. In this section we analyze how it impacts the gradients received by the gate. For simplicity, let us consider a simple transformer with a single attention layer and a class token at index 0.

### F.1. Without Adaptive Trimming.

The full process of MSViT can be summarized as:

#### 1. Obtain coarse level mask

$$\forall j \in [1, N_{S_c}], m_j = \text{GumbelSigmoid}(g_\psi(x_j)) \quad (11)$$

$$\bar{m}_j = \text{STE}(m_j) \quad (12)$$

#### 2. Deduce fine level mask

$$\forall i \in [N_{S_c} + 1, N_{S_c} + N_{S_f}], \bar{m}_i = 1 - \bar{m}_{C(i)} \quad (13)$$

#### 3. Embed patches and add position embeddings

#### 4. Masked attention

$$z_i = e^{Q_0 K_i^T} \quad (14)$$

$$y_0 = \sum_{i=1}^{N=N_{S_c}+N_{S_f}} \frac{\bar{m}_i z_i}{\sum_{p=1}^N \bar{m}_p z_p} V_i \quad (15)$$

where  $Q, K, V$  denotes the query, key and value embeddings of the tokens  $x'$ ; and  $C$  is the mapping from fine to coarse tokens.

#### 5. Feed $y_0$ to linear classification head

For simplicity, we will write the partition function as  $Z(\psi) = \frac{1}{\sum_{p=1}^N \bar{m}_p z_p}$ . Using the link between coarse and fine tokens from [Equation 13](#) we can decompose the equation in [step 4](#) as follows:

$$y_0 = Z(\psi) \sum_{i=1}^N \bar{m}_i z_i V_i \quad (16)$$

$$y_0 = Z(\psi) \left( \sum_{j=1}^{N_{S_c}} \bar{m}_j z_j V_j + \sum_{i=N_{S_c}+1}^N (1 - \bar{m}_{C(i)}) z_i V_i \right) \quad (17)$$

$$y_0 = Z(\psi) \left[ \underbrace{\sum_{j=1}^{N_{S_c}} \bar{m}_j \left( z_j V_j - \sum_{\substack{i=N_{S_c}+1 \\ C(i)=j}}^N z_i V_i \right)}_{A_j(\psi)} + \underbrace{\sum_{i=N_{S_c}+1}^N z_i V_i}_B \right] \quad (18)$$

Because of *straight-through*, we have  $\frac{\partial \bar{m}_j}{\partial \psi} = \frac{\partial m_j}{\partial \psi}$ , therefore every token contributes to the gradient with respect to the gate parameters,  $\frac{\partial y_0}{\partial \psi}$ , even if the token was masked with  $\bar{m}_j = 0$  in the forward pass. In particular, the fine and coarse tokens of each region directly interact through  $A_j(\psi)$ , where their attention value (wrt. the class token) are compared to each other.

### F.2. With Adaptive Trimming

With adaptive trimming, we reorder the tokens according to their value of  $\bar{m}_i$  and trim the number of tokens to the maximum sequence length in the batch in [step 4](#). This essentially mean that some terms will now be omitted from both the forward *and backward pass* in [Equation 15](#) and in [Equation 18](#). As a result, these terms also disappear from the gradients of  $Z(\psi)$  and  $A_j(\psi)$ . In particular, if the coarse token  $j$  is active and all corresponding fine tokens are trimmed, then:

$$\frac{\partial A_j(\psi)}{\partial \psi} = \frac{\partial m_j}{\partial \psi} z_j V_j \quad (19)$$

In the opposite case (fine scale active and corresponding coarse token is trimmed) then:

$$\frac{\partial A_j(\psi)}{\partial \psi} = -\frac{\partial m_j}{\partial \psi} \sum_{\substack{i=N_{S_c}+1 \\ C(i)=j}}^N z_i V_i \quad (20)$$

In other words, in the masking scenario ([Equation 18](#)) the transition from coarse to fine scale for a token is smoothly captured in the straight-through gradients  $\frac{\partial m_j}{\partial \psi}$ . In contrast, with adaptive trimming, flipping from coarse to fine tokens may sometimes lead to a sudden change in gradients from (19) to (20). Thus Adaptive trimming leads to a noisier optimization process. However, as we will see in the next section, this is not a significant issue in practice.

Model (ViT-S/16 backbone)		train time [min]	# tokens (average)	GMACs	Acc. [%]
ViT	in = 224	29.4	196	4.60	82.02
Mixed-Scale	$g^* = 0.5$ , <b>AT</b>	31.8	147	3.43	81.53
	$g^* = 0.5$ , <b>full</b>	36.0	155	3.62	81.71
	$g^* = 0.1$ , <b>AT</b>	28.8	117	2.73	80.63
	$g^* = 0.1$ , <b>full</b>	36.0	132	3.09	80.96

Table 5. Average training time per epoch (in minutes) for our mixed-scale MSViT-S/16, with (**AT**) and without (**full**) adaptive trimming during training. In practice, ATP leads to faster training time, and only a minor drop in accuracy for comparable MAC count. We also report the original ViT backbone timings for reference.

### F.3. Adaptive trimming in practice

In Table 5, we report a comparison of training times for MSViT, with and without the adaptive token trimming (AT) strategy introduced in Section 2.3. As expected, AT leads to faster training times, in particular for lower values of the gate sparsity target  $g^*$ . Furthermore, in practice we observe that AT generally yields comparable trade-off (or only incurs a minor drop in accuracy for comparable MAC counts), which is why we make it the default for all training runs in our main experiments.

## G. Additional ablation experiments

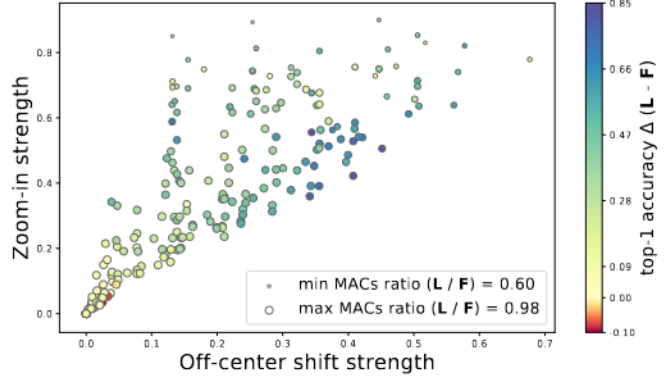
### G.1. Benefits of a dynamic gate

As described in Section 4.3.2, the learned dynamic gate in MSViT is able to adapt the model’s computational cost based on the input image content, in contrast to using a fixed mixed-scale pattern. This is illustrated in Figure 14: Here, we generate several random geometric shifts of the validation set, and evaluate both a fixed and learned gate model. We then report in Figure 14 (b) their difference in accuracy (color of the scatter plot) and in MAC counts (size of the scatter plot). We observe that:

- (i) The learned gate model generally outperforms the fixed gate one and this is more pronounced when the random transformation has a strong off-center shift; In fact, in those cases, the prior of the fixed gate that objects lie in the center of the image breaks.
- (ii) The fixed scale selection pattern leads to computational waste when applied on these simple affine geometric shifts that mimic more realistic “in-the-wild” image inputs. In fact the computational cost of the fixed gate model is constant; while the cost of the learned gate model is significantly reduced when we have strong shifts, as they generally lead to more background regions present in the image, as visualized in Figure 14 (a).



(a) Example gate outputs given random image zooms and shifts.



(b) Each point corresponds to a different cropping transform applied to all images of the validation set, and both models have similar starting performances (83.17 accuracy at 8.50GMACs for **L**; 83.06 at 8.75GMACs for **F**). The colors encode accuracy improvement of **L** compared to **F** (the bluer the better), and the dot size encodes the efficiency improvement (the smaller the better) of the learned gate over the fixed gate.

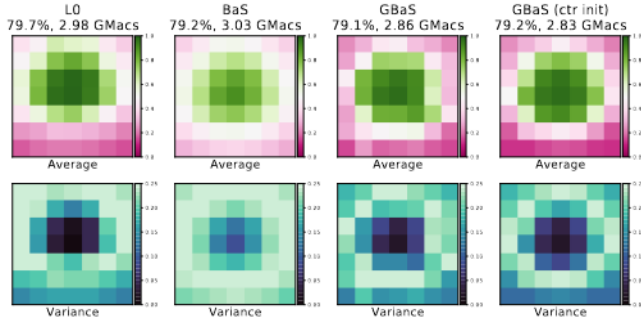
Figure 14. Performance of a learned gate model (**L**) versus a fixed radial masking pattern (**F**). We observe that in most scenarios **L** provides better accuracy and automatically adapts its computational cost accordingly: For instance, highly zoomed-in images tend to contain more background/flat color patches, which are set to coarse scale by the learned gate, leading to lower MACs.

### G.2. Generalized Batch Shaping loss

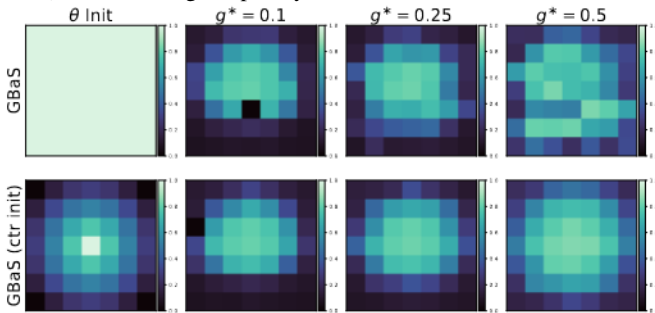
In Figure 15, we report the counterpart of Figure 7 for light croppings data augmentations. As mentioned in the main text, in that setting, there is little to no shift in spatial distribution between train and test time. As a result, all gate losses manage to capture the prior that objects tend to lie in the center of the image in ImageNet (see Figure 15 (a)). Similarly, for GBaS, even without dedicated initialization the learned priors also fit the central locality insight (Figure 15 (b)). All losses perform similarly in that setting, and the fast-converging L0 loss is even able to outperform BaS and GBaS in Figure 6 (b).

### G.3. Rescaling the position embeddings with linear interpolation

In Figure 16 we show that, when using the standard ViT finetuning pipeline with the linear interpolation of position encodings leads to an interesting observation: For a low number of tokens, ViT-S/32 on image size  $X$  (scenario **A**) performs better than a ViT-S/16 model on image size  $X//2$



(a) Average (top row) and variance (bottom row) of the learned scale selection masks across the validation set (A value above 0.5 means that the corresponding image patch will be kept at fine scale) for different gate sparsity losses.



(b) Prior parameters  $\theta$  learned with the GBaS loss with/without ctr init (top/bottom). The first column is initial values of  $\theta$ . Figure 15. Illustration of the masks learned by the model with light crops data augmentation, leading to little to no shift between the train and test distribution of the tokens input to the gate

(scenario B), despite them having the same number of tokens.

We then investigate whether this behavior also occurs in MSViT. In Figure 17, we report results for the setting described in the main paper: ViT-S/16 backbone at different input image sizes, and MSViT-S/{16, 32} for different gate loss objectives. In addition, we also report results on ViT-S/32 and MSViT-S/{32, 64}, run on a subset of the search space.

As we see from Figure 17, the impact of patch size is in fact the same as in ViT: In the low regime of the number of tokens (around 95), MSViT-S/32, 64 ran on larger images starts to outperform ViT-S/16, 32. This also indicates that the token embedding and resizing algorithm may impact the model's accuracy in for a low number of tokens, and motivates further investigation of this behavior for vision transformers in general.

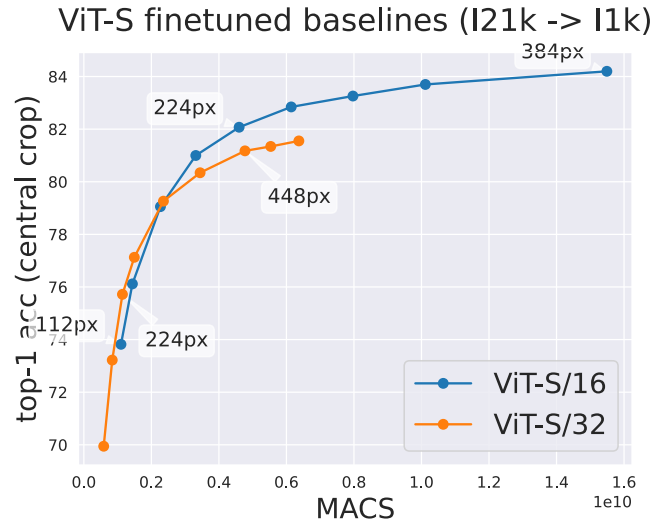


Figure 16. Comparison of the performance of ViT-S with patch size 32 and patch size 16, trained for different input image sizes using the linear interpolation rescaling trick of the position embeddings. While ViT-S/16 generally yields better trade-offs, the trend starts to invert itself around the threshold of 100 tokens

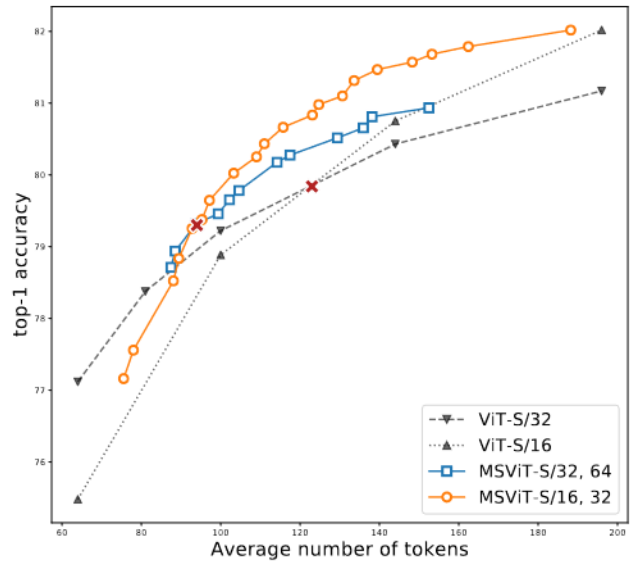


Figure 17. Comparing the effect of patch scale versus input image size: In terms of number of tokens, increasing the patch or decreasing the input image size are equivalent; However, the initial token embeddings and resizing differ in these two settings; As we can see from this plot, this can lead to large differences in the low token regimes for the standard ViT backbone ( $\sim 130$  tokens, indicated by X), and we see the same shift starting to appear for MSViT models around 90 tokens.