Token Pooling in Vision Transformers for Image Classification

Supplementary Material

Dmitrii Marin†, Jen-Hao Rick Chang⋆, Anurag Ranjan⋆, Anish Prabhu⋆
Mohammad Rastegari⋆, Oncel Tuzel⋆
†University of Waterloo, ⋆Apple
jenhao_chang@apple.com

A. Weighted clustering algorithms

Weighted K-Means minimizes the following objective w.r.t. $\hat{\mathcal{F}} = \{\hat{\boldsymbol{f}}_1, \ldots, \hat{\boldsymbol{f}}_K\} \subset \mathbb{R}^M$:

$$\ell(\mathcal{F}, \hat{\mathcal{F}}) = \sum_{\boldsymbol{f}_i \in F} \min_{\hat{\boldsymbol{f}}_j \in \hat{F}} w_i \|\boldsymbol{f}_i - \hat{\boldsymbol{f}}_j\|^2 \tag{13}$$

The extension of the K-Means algorithm to the weighted case iterates the following steps:

$$a(i) \leftarrow \arg\min_j \|\boldsymbol{f}_i - \hat{\boldsymbol{f}}_j\| \qquad \forall i \in \{1, 2, \ldots, N\}, \tag{14}$$

$$\hat{\boldsymbol{f}}_j \leftarrow \frac{\sum_{i=1}^N [a(i) = j] w_i \boldsymbol{f}_i}{\sum_{i=1}^N [a(i) = j] w_i} \qquad \forall j \in \{1, 2, \ldots, K\}., \tag{15}$$

Weighted K-Medoids optimizes objective (13) under the medoid constraint $\hat{\mathcal{F}} \subset \mathcal{F}$:

$$a(i) \leftarrow \arg\min_j \|\boldsymbol{f}_i - \hat{\boldsymbol{f}}_j\| \qquad \forall i \in \{1, 2, \ldots, N\}, \tag{16}$$

$$n(j) \leftarrow \arg\min_{i:\, z(i)=j} \sum_{i':\, a(i')=j} \|\boldsymbol{f}_i - \boldsymbol{f}_{i'}\|^2 \qquad \forall j \in \{1, 2, \ldots, K\}, \tag{17}$$

$$\hat{\boldsymbol{f}}_j \leftarrow \boldsymbol{f}_{n(j)}. \tag{18}$$

B. Ablations on clustering initialization

We examine the effect of the cluster center initialization. We compare our default initialization, which uses the tokens with top-K significance scores as initial cluster centers, with random initialization, which randomly selects tokens as initial cluster centers. As shown in Figure 7, Token Pooling is robust to the initialization methods.

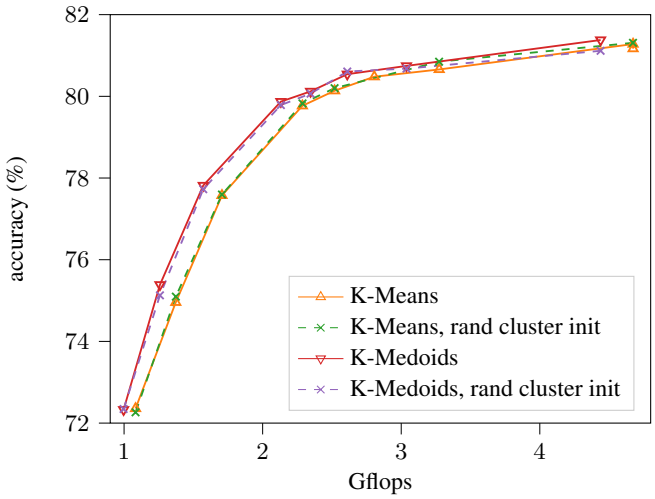


Figure 7: Default initialization *vs.* random initialization. Our Token Pooling is robust to initialization of clustering algorithms. The default initialization is top-K w.r.t. significance score, see Algorithm 1.

C. Training details & hyper-parameters

A Token Pooling layer has one parameter — the number of the output tokens K (or equivalently the downsampling ratio) — which is the same as downsampling layers like max or average pooling. For max or average pooling layers, the sizes of the kernels and strides play the same role as K. We use the method proposed by Goyal *et al*. [2] to automatically select Ks of all Token Pooling layers (see below for details). All other hyperparameters are inherited from DeiT [9].

To fairly compare PoWER-BERT and the baseline methods with the proposed Token Pooling, all methods (except convolution downsampling) use the same target number of tokens for downsampling layers after each transformer block. Specifically, we run the second stage of the PoWER-BERT training [2] for 30 epochs with various values of the token-selection weight parameter λ producing a family of models. The single parameter λ controls the overall efficiency of a model, and the numbers of retained tokens of all Token Pooling layers are selected automatically based on the choice of λ. Each of the resulted models has a different number of retained tokens at each of its L transformer blocks: $\boldsymbol{K} = (K_1, \ldots, K_L)$. Appendix G lists all combinations of automatically determined $\boldsymbol{K}$. We then finetune these models using the third (last) stage of PoWER-BERT. Note that we apply the same process for PoWER-BERT, random selection, importance selection, and our Token Pooling, and we use the same $\boldsymbol{K}$ when comparing these methods.

We find that the DeiT models provided by Touvron *et al*. [9] are under-fit, and their accuracy improves with additional training, see Figure 8. After the standard DeiT training, we restart the training. This ensures that downsampling models and DeiT with a similar number of training steps.

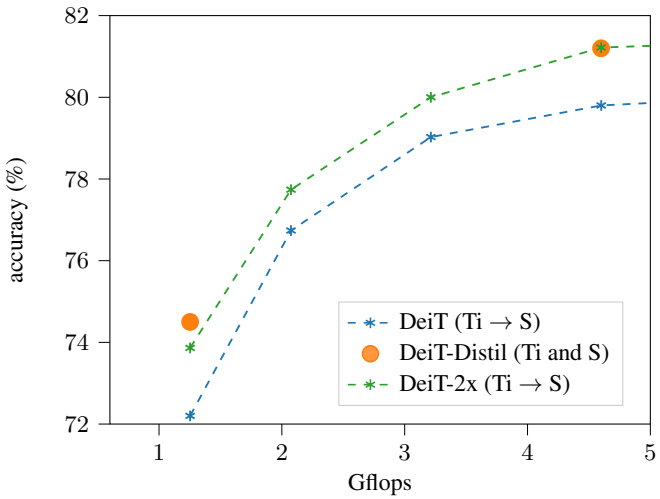


Figure 8: This figure shows the results of the pretrained DeiT models provided by Touvron *et al*. [9] (*DeiT*) and the DeiT models trained with our protocol (*DeiT-2x*). Our training protocal uses the same hyper-parameters provided by Touvron *et al*. [9], but after the model is trained, we finetune the model using the same hyper-parameters (*i.e*., restart the learning rate schedule). We also show *DeiT-Distil* results (cited from [9]), which use knowledge distillation.

D. Convolution downsampling (generalized average pooling)

As mentioned in the main paper, we enumerate combinations of layers to insert the convolution downsampling layer. We use 2x2 convolution with stride 2 — the same operation as the Patch Merging layer used by Swin [5]. To keep the feature dimensionality of DeiT the same (and evaluate the pure effect of the downsampling layers), the output feature dimensionality is the same as the input. With 196 tokens in DeiT-S model, we can include no more than 3 convolution downsampling layers as each layer reduces the number of token by a factor of 4. When using 3 layers at depths $l_1 < l_2 < l_3$, we restrict $l_2 - l_1 = l_3 - l_2$. With these constraints, we enumerate all possible downsampling configurations, *i.e*., 1 and 2 layers and 3 layers satisfying the condition. Each of the combinations produces a model with a different computation-accuracy trade-off, and we report the Pareto front, *i.e*., the best accuracy these models achieve at a given flop. See Figure 9.

Note that the convolution downsampling layer is equivalent to the patch-merging layer used by Liu *et al*. [5], Heo *et al*. [3], and Wang *et al*. [11] (except that we keep the output feature dimension the same to compare fairly with DeiT). It is also a generalized version of the commonly used average pooling layer (*e.g*., used by Roy *et al*. [7]). Average pooling can be implemented by convolving the feature map with a constant kernel of size $n \times n$ containing $\frac{1}{n^2}$ and subsample (stride) every

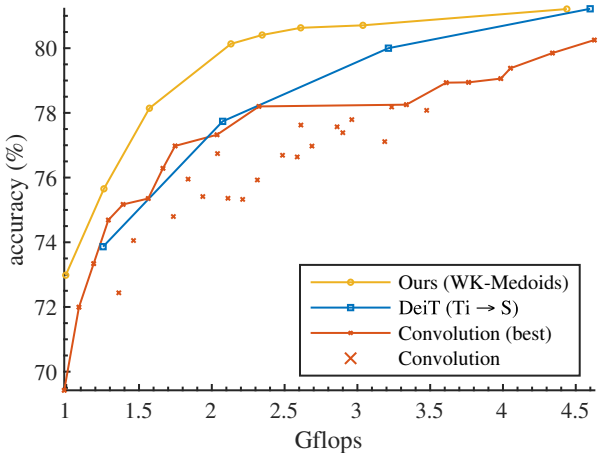


Figure 9: Results of convolution downsampling

n pixels. By learning its kernel, the convolution downsampling layer is able to find a kernel suitable for the downsampling and the current task than average pooling.

E. Directly inserting Token Pooling into pretrained transformers

In this section, we directly insert Token Pooling layers into a *pre-trained* model. While such operation (modifying network architecture post training) is unusual and typically degrades performance dramatically, this experiment enables us to evaluate how well Token Pooling preserving information during the downsampling.

Figure 10 shows the results when we directly insert Token Pooling layers (using the same downsampling ratios in Figure 6b) into a pretrained DeiT-S. As can seen, even though Token Pooling layers are added *post training*, we still can reduce a small number of tokens without significant degradation in accuracy. The result indicates that Token Pooling preserves information that enables the model to retain accuracy during token downsampling.

In Figure 10, we also show the results of an alternative method that better preserve information of pre-trained transformers (WK-Means, carry and WK-Medoids, carry). Specifically, in addition to outputting K cluster centers, we count the number of tokens assigned to a cluster center and *carry* the count when we compute softmax-attention in the next transformer blocks. This operation preserves the attention weights, and since the models are not trained after inserting Token Pooling layers, it preserves more information. In practice, we observe that when the models are trained with Token Pooling layers, the carry operation is not important.

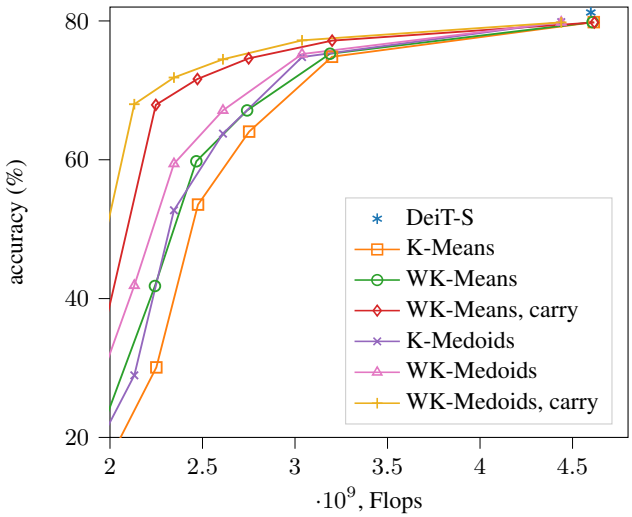
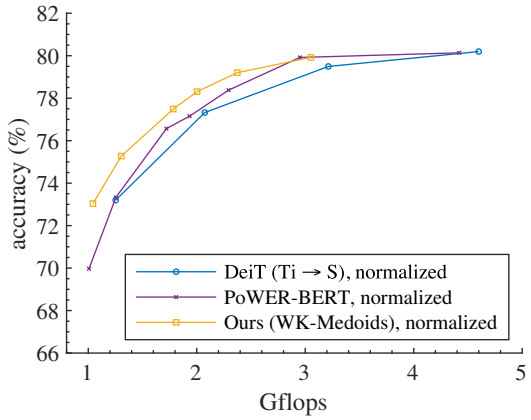
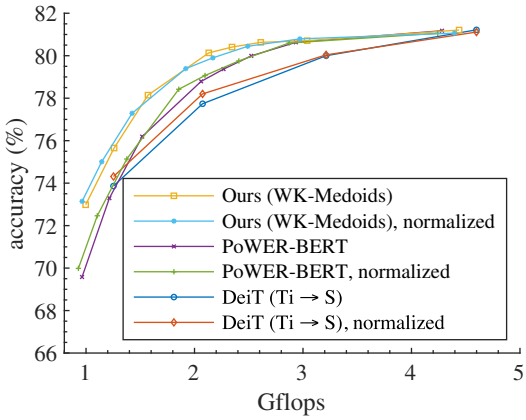


Figure 10: The figure shows the performance results when we directly insert Token Pooling layers into a *pretrained* DeiT-S.



(a) $\alpha = 1$



(b) Learned α for each layer

Figure 11: Results of models using normalized key and query vectors with (a) $\alpha = 1$ and (b) learned α in (6). The base model architecture is DeiT-S.

F. Normalized key and query vectors

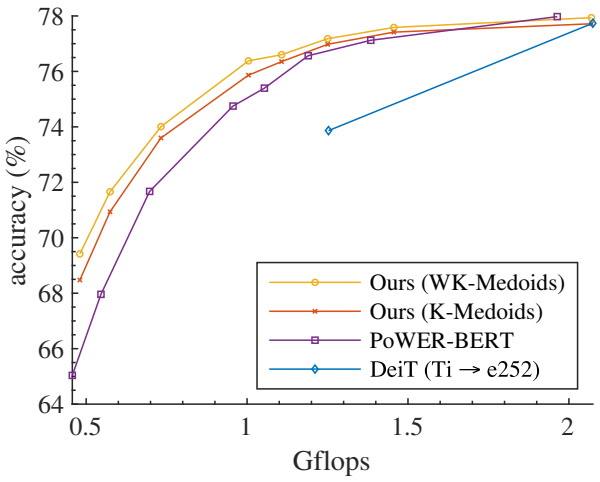
Our analysis of softmax-attention in Section 3.3 assumes the query and the key vectors are normalized to have a constant ℓ^2 norm. It has been observed by Kitaev *et al*. [4] that normalizing key and query vectors does not change the performance of a transformer. Thus, for all experiments, we train models *without* the norm normalization. In this section, we verify this observation by training various DeiT, PoWER-BERT, and Token Pooling *with* normalized keys and queries. We found that the scalar α in the softmax attention layer (6) can affect the performance of a transformer with normalized keys and queries. As can be seen in Figure 11a, setting the scalar $\alpha = 1$ slightly deteriorates the performance of the model. Instead of using a fixed α, we let the model learn the α for each layer. As can be seen in Figure 11b, learning α enables the resulting models to achieve similar cost-accuracy trade-off as the standard (unnormalized) models. With or without the normalization and the learnable α, the proposed Token Pooling significantly improves the cost-accuracy trade-off of DeiT and outperforms PoWER-BERT.

G. Ablations of clustering algorithms

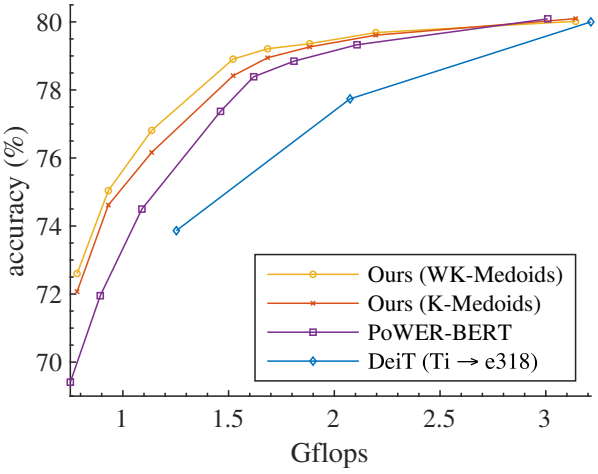
Tables 2–5 detail the results of PoWER-BERT and Token Pooling on the DeiT architectures that we tested (DeiT-S, DeiT-e318, and DeiT-e252). Figure 12 shows the ablation results with different backbone models. Table 2 details the results of the best cost-accuracy trade-off achieved by the proposed Token Pooling (using K-Medoids and WK-Medoids) and PoWER-BERT via varying token sparsity and feature dimensionality of DeiT.

Apart from the standard K-Means and K-Medoids, other clustering approaches could be used. Many methods are not suitable due to efficiency constraints. For example, normalized cut [8] uses expensive spectral methods. One prerequisite of using K-Means and K-Medoids is the number of clusters, K. While selecting K for each of the layers may be tedious and difficult, one can choose K via computational budgets and heuristics. In this work, we use the automatic search procedure proposed by Goyal *et al*. [2] to determine K, see Appendix C.

Since both K-Means and K-Medoids require specifying the number of clusters K in advance, one may consider using methods automatically determining K. Nevertheless, such methods typically have other parameters, which are less interpretable than K. For example, mean-shift [1] or quick-shift [10] require specifying the kernel size. From our experience, determining these parameters is challenging. In addition, since the number of clusters (and hence the number of output tokens) is determined on the fly during inference, the computational requirement can fluctuate, making deployment of these models difficult.



(a) DeiT-e252 as the base architecture



(b) DeiT-e318 as the base architecture

Figure 12: This figure compares the cost-accuracy curves of Token Pooling with PoWER-BERT using (a) DeiT-e252 and (b) DeiT-e318 as the base architectures.

DeiT (Ti → S)		PoWER-BERT		Token Pooling (K-Medoids)		Token Pooling (WK-Medoids)	
Gflops	Accuracy (%)	Gflops	Accuracy (%)	Gflops	Accuracy (%)	Gflops	Accuracy (%)
-	-	0.46	65.0	0.48	68.5	0.48	**69.4**
-	-	0.55	68.0	0.57	70.9	0.57	**71.7**
-	-	0.70	71.7	0.73	73.6	0.73	**74.0**
-	-	0.89	72.0	0.93	74.6	0.93	**75.0**
-	-	0.96	74.8	1.00	75.9	1.00	**76.4**
-	-	1.05	75.4	1.11	76.4	1.11	**76.6**
1.25	73.9	1.19	76.6	1.25	77.0	1.25	**77.2**
-	-	1.38	77.1	1.46	77.4	1.46	**77.6**
-	-	1.46	77.4	1.52	78.4	1.52	**78.9**
-	-	1.62	78.4	1.68	78.9	1.68	**79.2**
2.07	77.7	1.81	78.8	1.88	79.3	1.88	**79.4**
-	-	2.11	79.3	2.13	79.9	2.13	**80.1**
-	-	2.27	79.4	2.35	80.1	2.35	**80.4**
-	-	2.52	80.0	2.61	80.5	2.61	**80.6**
3.21	80.0	2.93	80.6	3.04	**80.7**	3.04	**80.7**
4.60	81.2	4.28	81.2	4.44	**81.4**	4.44	81.2

Table 2: Best cost-accuracy trade-off achieved by PoWER-BERT and the proposed Token Pooling via varying sparsity level and feature dimensionality.

clustering method	Weighting	ImageNet Accuracy	GFlops
Sparsity level 0: $\boldsymbol{K} = [196, 196, 195, 194, 189, 180, 173, 173, 173, 173, 173, 173]$			
PoWER-BERT	N/A	81.2	4.3
K-Means		81.3	4.7 (+0.4)
	✓	81.1	4.7 (+0.4)
K-Medoids		**81.4**	4.4 (+0.1)
	✓	81.2	4.4 (+0.1)
Sparsity level 1: $\boldsymbol{K} = [196, 195, 193, 188, 169, 140, 121, 110, 73, 38, 7, 0]$			
PoWER-BERT	N/A	80.6	2.9
K-Means		80.7	3.3 (+0.4)
	✓	**80.8**	3.3 (+0.4)
K-Medoids		80.7	3.0 (+0.1)
	✓	80.7	3.0 (+0.1)
Sparsity level 2: $\boldsymbol{K} = [196, 195, 190, 177, 141, 108, 84, 69, 35, 18, 3, 0]$			
PoWER-BERT	N/A	80.0	2.5
K-Means		80.5	2.8 (+0.3)
	✓	80.5	2.8 (+0.3)
K-Medoids		80.5	2.6 (+0.1)
	✓	**80.6**	2.6 (+0.1)
Sparsity level 3: $\boldsymbol{K} = [196, 194, 187, 163, 118, 85, 58, 47, 20, 12, 2, 0]$			
PoWER-BERT	N/A	79.4	2.3
K-Means		80.1	2.5 (+0.2)
	✓	**80.4**	2.5 (+0.2)
K-Medoids		80.1	2.3 (+0.08)
	✓	**80.4**	2.3 (+0.08)
Sparsity level 4: $\boldsymbol{K} = [196, 193, 179, 142, 97, 64, 46, 34, 13, 9, 1, 0]$			
PoWER-BERT	N/A	78.8	2.1
K-Means		79.8	2.3 (+0.2)
	✓	80.0	2.3 (+0.2)
K-Medoids		79.9	2.1 (+0.07)
	✓	**80.1**	2.1 (+0.07)
Sparsity level 5: $\boldsymbol{K} = [194, 183, 142, 89, 41, 20, 10, 7, 0, 0, 0, 0]$			
PoWER-BERT	N/A	76.2	1.5
K-Means		77.6	1.7 (+0.2)
	✓	**78.1**	1.7 (+0.2)
K-Medoids		77.8	1.6 (+0.05)
	✓	**78.1**	1.6 (+0.05)
Sparsity level 6: $\boldsymbol{K} = [186, 162, 102, 56, 13, 4, 2, 2, 0, 0, 0, 0]$			
PoWER-BERT	N/A	73.3	1.2
K-Means		75.0	1.4 (+0.2)
	✓	75.6	1.4 (+0.2)
K-Medoids		75.4	1.3 (+0.04)
	✓	**75.7**	1.3 (+0.04)
Sparsity level 7: $\boldsymbol{K} = [162, 129, 66, 33, 4, 1, 1, 0, 0, 0, 0, 0]$			
PoWER-BERT	N/A	69.6	1.0
K-Means		72.4	1.1 (+0.1)
	✓	**73.0**	1.1 (+0.1)
K-Medoids		72.3	1.0 (+0.03)
	✓	**73.0**	1.0 (+0.03)

Table 3: Results of applying Token Pooling and PoWER-BERT on **DeiT-S** model. The models are grouped by $\boldsymbol{K}$ described in Appendix C. The integer list denotes the maximal number of tokens retained after each transformer block. These numbers do not take into account the classification token, which is always retained. Thus, "0" means that only the classification token remains. Additional flops (denoted by the parentheses) are due to clustering.

clustering method	Weighting	ImageNet Accuracy	GFlops
Sparsity level 0: $\boldsymbol{K} = [196, 196, 196, 194, 192, 184, 181, 181, 170, 170, 170, 170]$			
PoWER-BERT	N/A	80.1	3.0
K-Medoids		**80.1**	3.1 (+0.1)
	✓	80.0	3.1 (+0.1)
Sparsity level 1: $\boldsymbol{K} = [196, 195, 195, 190, 171, 147, 132, 122, 66, 43, 15, 0]$			
PoWER-BERT	N/A	79.3	2.1
K-Medoids		79.6	2.2 (+0.1)
	✓	**79.7**	2.2 (+0.1)
Sparsity level 2: $\boldsymbol{K} = [196, 195, 193, 180, 143, 109, 92, 77, 40, 21, 4, 0]$			
PoWER-BERT	N/A	78.8	1.8
K-Medoids		79.3	1.9 (+0.09)
	✓	**79.4**	1.9 (+0.09)
Sparsity level 3: $\boldsymbol{K} = [196, 195, 190, 165, 119, 84, 65, 50, 26, 14, 3, 0]$			
PoWER-BERT	N/A	78.3	1.6
K-Medoids		78.9	1.7 (+0.07)
	✓	**79.2**	1.7 (+0.07)
Sparsity level 4: $\boldsymbol{K} = [196, 194, 184, 149, 97, 65, 47, 33, 12, 9, 2, 0]$			
PoWER-BERT	N/A	77.4	1.5
K-Medoids		78.4	1.5 (+0.06)
	✓	**78.9**	1.5 (+0.06)
Sparsity level 5: $\boldsymbol{K} = [196, 186, 153, 93, 40, 15, 12, 9, 0, 0, 0, 0]$			
PoWER-BERT	N/A	74.5	1.1
K-Medoids		76.2	1.1 (+0.05)
	✓	**76.8**	1.1 (+0.05)
Sparsity level 6: $\boldsymbol{K} = [193, 173, 109, 52, 16, 4, 4, 4, 0, 0, 0, 0]$			
PoWER-BERT	N/A	72.0	0.89
K-Medoids		74.6	0.93 (+0.04)
	✓	**75.0**	0.93 (+0.04)
Sparsity level 7: $\boldsymbol{K} = [183, 145, 80, 33, 5, 1, 1, 2, 0, 0, 0, 0]$			
PoWER-BERT	N/A	69.4	0.75
K-Medoids		72.1	0.78 (+0.03)
	✓	**72.6**	0.78 (+0.03)

Table 4: Results of applying Token Pooling and PoWER-BERT on **DeiT-e318** model. The models are grouped by $\boldsymbol{K}$ described in Appendix C. The integer list denotes the maximal number of tokens retained after each transformer block. These numbers do not take into account the classification token, which is always retained. Thus, "0" means that only the classification token remains. Additional flops (denoted by the parentheses) are due to clustering.

clustering method	Weighting	ImageNet Accuracy	GFlops
Sparsity level 0: $\boldsymbol{K} = [196, 196, 196, 195, 193, 189, 177, 177, 177, 177, 177, 177]$			
PoWER-BERT	N/A	**78.0**	2.0
K-Medoids		77.7	2.1 (+0.1)
	✓	77.9	2.1 (+0.1)
Sparsity level 1: $\boldsymbol{K} = [196, 196, 195, 189, 176, 161, 123, 122, 76, 48, 17, 0]$			
PoWER-BERT	N/A	77.1	1.4
K-Medoids		77.4	1.5 (+0.07)
	✓	**77.6**	1.5 (+0.07)
Sparsity level 2: $\boldsymbol{K} = [196, 196, 193, 179, 154, 123, 88, 79, 39, 22, 5, 0]$,			
PoWER-BERT	N/A	76.6	1.2
K-Medoids		77.0	1.3 (+0.06)
	✓	**77.2**	1.3 (+0.06)
Sparsity level 3: $\boldsymbol{K} = [196, 195, 188, 167, 131, 96, 63, 52, 13, 11, 2, 0]$,			
PoWER-BERT	N/A	75.4	1.1
K-Medoids		76.4	1.1 (+0.05)
	✓	**76.6**	1.1 (+0.05)
Sparsity level 4: $\boldsymbol{K} = [196, 194, 181, 147, 111, 75, 48, 36, 5, 7, 2, 0]$			
PoWER-BERT	N/A	74.8	0.96
K-Medoids		75.9	1.0 (+0.04)
	✓	**76.4**	1.0 (+0.04)
Sparsity level 5: $\boldsymbol{K} = [196, 187, 129, 88, 54, 20, 10, 10, 0, 1, 0, 0]$			
PoWER-BERT	N/A	71.7	0.70
K-Medoids		73.6	0.73 (+0.03)
	✓	**74.0**	0.73 (+0.03)
Sparsity level 6: $\boldsymbol{K} = [194, 163, 83, 44, 22, 6, 1, 3, 0, 0, 0, 0]$			
PoWER-BERT	N/A	68.0	0.55
K-Medoids		70.9	0.57 (+0.02)
	✓	**71.7**	0.57 (+0.02)
Sparsity level 7: $\boldsymbol{K} = [188, 122, 58, 28, 12, 2, 0, 2, 0, 0, 0, 0]$			
PoWER-BERT	N/A	65.0	0.46
K-Medoids		68.5	0.48 (+0.02)
	✓	**69.4**	0.48 (+0.02)

Table 5: Results of applying Token Pooling and PoWER-BERT on **DeiT-e252** model. The models are grouped by $\boldsymbol{K}$ described in Appendix C. The integer list denotes the maximal number of tokens retained after each transformer block. These numbers do not take into account the classification token, which is always retained. Thus, "0" means that only the classification token remains. Additional flops (denoted by the parentheses) are due to clustering.

H. Throughput

Throughput reflects the actual speed during inference time, and it is often measured by the number of processed images per second (fps). As a result, the value of throughput highly depends on the specific hardware, implementation quality, and the workload and the state (*e.g.*, temperature) of the machines [6]. Since it is usually difficult to control all these factors to have a fair comparison with other papers, in the main paper, we choose to report the theoretical computational cost (*i.e.*, flops) that is known to be highly correlated with the energy consumption on device.

In Figure 13, we provide the throughput of our models, served as a reference on how our models actually perform given our implementation and the type of GPU used. Note that our PyTorch implementation of the clustering algorithms can be significantly improved, for example, via implementing as a CUDA kernel. To determine the throughput, we run the model inference several times with different batch sizes. We report the average throughput of 30 different runs using the best batch size. As can be seen, with the same accuracy, using Token Pooling increases the throughput of DeiT-e318 by 25% (1640 *vs.* 1310 fps), of DeiT-e252 by 22% (2190 *vs.* 1791 fps), and of DeiT-Ti by 14% (3220 *vs.* 2834 fps).

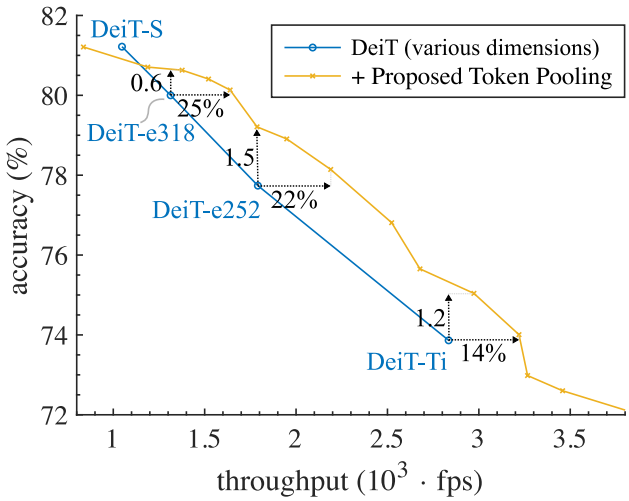


Figure 13: Throughput of our PyTorch implementation. Note that our implementation has not been optimized for the throughput. The throughput is measured in frames (images) per second (fps). These numbers are measured on a single Nvidia V100 GPU. As can be seen, despite our non-optimized code, Token Pooling significantly improves both the flops (see Figure 1a) and throughput of DeiT models.

References

[1] Yizong Cheng. Mean shift, mode seeking, and clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 17(8):790–799, 1995.

[2] Saurabh Goyal, Anamitra Roy Choudhury, Saurabh Raje, Venkatesan Chakaravarthy, Yogish Sabharwal, and Ashish Verma. PoWER-BERT: Accelerating BERT inference via progressive word-vector elimination. In *International Conference on Machine Learning (ICML)*, pages 3690–3699, 2020.

[3] Byeongho Heo, Sangdoo Yun, Dongyoon Han, Sanghyuk Chun, Junsuk Choe, and Seong Joon Oh. Rethinking spatial dimensions of vision transformers. In *IEEE International Conference on Computer Vision (ICCV)*, 2021.

[4] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations (ICLR)*, 2020.

[5] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows, October 2021.

[6] Danny Price, Ben Barsdell, Lincoln Greenhill, Mike Clark, and Ron Babich. Fire and ice: How temperature affects gpu performance. In *NVIDIA GTC*. 2014.

[7] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

[8] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 22(8):888–905, 2000.

[9] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. pages 10347–10357, 2021.

[10] Andrea Vedaldi and Stefano Soatto. Quick shift and kernel methods for mode seeking. In *European Conference on Computer Vision (ECCV)*, pages 705–718, 2008.

[11] Wenhai Wang, Enze Xie, Xiang Li, Deng-Ping Fan, Kaitao Song, Ding Liang, Tong Lu, Ping Luo, and Ling Shao. Pyramid vision transformer: A versatile backbone for dense prediction without convolutions. In *IEEE International Conference on Computer Vision (ICCV)*, pages 568–578, October 2021.