A APPENDIX

Here, we will demonstrate the additional results from our work.

A.1 SOBEL OPERATOR

The Sobel operator calculates the gradient of an image by convolving the image with two 3x3 kernels: one for detecting horizontal edges (G_x) and one for detecting vertical edges (G_y). The convolution of the image I at each pixel (i, j) with these kernels is expressed as:

$$G_x(i,j) = \sum_{m=-1}^{1} \sum_{n=-1}^{1} G_x(m,n) \cdot I(i+m, j+n) \tag{6}$$

$$G_y(i,j) = \sum_{m=-1}^{1} \sum_{n=-1}^{1} G_y(m,n) \cdot I(i+m, j+n) \tag{7}$$

where the Sobel kernels are defined as:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{8}$$

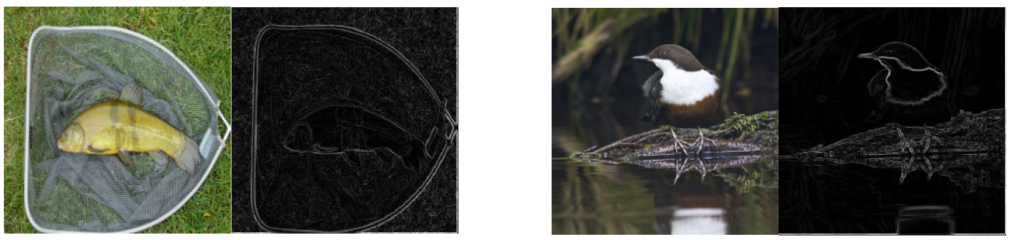
The gradient magnitude is then calculated as:

$$\text{Magnitude}(i,j) = \sqrt{G_x(i,j)^2 + G_y(i,j)^2} \tag{9}$$

The gradient magnitude is a measure of the sharpness of the image - a measure of how much curvature or information is there in the image.

A.2 VISUAL REPRESENTATION OF SIMPLE AND COMPLEX IMAGES

Figure 7 is a visual representation of a simple and a complex image from the ImageNet dataset. We identify few curves on the simple image and more detailed silhouettes on complex images, based on our thesholding mechanism in Section 3.1.



(a) Simple Image Thresholding **(b) Complex Image Thresholding**

Figure 7: Examples from the ImageNet dataset (12) illustrating the application of a Sobel operator [A.1] for edge detection. (a) A "simple" image (left) with minimal details, processed with the Sobel operator (right), showing fewer prominent edges. (b) A "complex" image (right) with intricate details, processed with a Sobel operator (right), revealing a dense edge pattern.

A.3 SPEED-UP DUE TO ATTENTION PARALLELIZATION

As discussed in Section 3.3.3, parallelizing attention computations significantly accelerates inference in the Efficient Early-Exit Transformer, with a more pronounced effect on 2D tasks than 3D. In 3D tasks, the extra pre-processing for point cloud data and 2D projection introduces additional computational overhead, limiting the speedup.

Table 6: **Parallel attention computation speed-up:** Comparison of data instances processed per second for Early-Exit Efficient Transformer, evaluated under both parallel and sequential attention schemes. The speedup in 2D images is more than 3D shapes.

Dataset	Data instances processed per second	
	Attention parallel	Attention sequential
CIFAR 100 (24)	60.15	58.37
ImageNet (12)	28.23	26.40
ModelNet40 (64)	13.74	13.57

A.4 Comparison with extremely small models

Despite the existence of models with considerably lower MACs in Table 7, our evaluation reveals a clear advantage for our approach. Our model exhibits faster inference times, notably on edge devices, and maintains a higher accuracy than any of the identified low-MAC models. These experiments have been performed on ImageNet. We also add entries from recent models whose implementation is not publicly available at the time of submission. We use the symbol: − to indicate observations that are not accessible due to the aforementioned reason.

Table 7: **Comparison with other models:** Comparison of small MAC Methods with UWYN on ImageNet. We demonstrate higher accuracy than all and faster inference for some.

Method	MACs	Params	Accuracy	Time (Xavier)	Time (P100)	Time (Orin)
Efficient ViT-M0 (36)	0.1 G	5.4 M	71.9%	1415 s	346.87 s	1279.38 s
LeViT-128S (14; 59)	0.2 G	7.77 M	76.5%	1582.14 s	401.16 s	1632.71 s
EfficientFormerV2-s0 (29; 59)	0.3 G	3.5 M	76.1%	1994.42 s	494.13 s	2068.05 s
LF-ViT (17)	1.85 G	-	82.2%	-	-	-
CF-ViT (7)	2.4 G	-	81.9%	-	-	-
SAC-ViT (18)	1.6 G	-	82.3%	-	-	-
UWYN	1.2 G	22.86 M	84.39%	1796 s	492.63 s	1695.23 s

A.5 Using a larger batch size to compare results

Here we compare the inference time when we use a batch size of 32 during inference time. There is a larger speedup in the 3D pre-processing due to efficient handling of data as compared to the other methods. We identify the simple and complex data instances from beforehand, batch them together and then perform inference.

Figure 8: **Batch inference:** This table shows the time required for inference using a batch size of 32 on the ModelNet40 dataset.

Method	Inference Time
PointNet++ (45)	695.45 sec
Point Transformer (72)	701.88 sec
UWYN	166.39 sec

Figure 9: **Batch inference:** This table shows the time required for inference using a batch size of 32 on the CIFAR 10 dataset.

Method	Inference Time
LGViT (66)	59.83 sec
AdaptFormer (8)	62.18 sec
UWYN	40.45 sec

A.6 MACs for other networks

In this section we will compare the MACs of our networks with other popular transformer architectures. We are using the maximum MAC for our methods for comparison. From the table below, we demonstrate that our MACs are minimal with respect to other architectures as well.

Table 8: **MACs of other methods:** From the table, it is evident that our method is much more computationally efficient than the existing popular architecture choices. We use the thop (5) python library to calculate the MACs. Our MACs are very low compared to the other state of the art.

Architecture	MACs
DeIT Small (55; 62; 12)	4249 M
Swin V2 Tiny (37)	5760 M
Mobile ViT small (40)	347.5 M
EfficientNet-B0 (50)	380.55 M
ConvNeXt-T (13)	4.5 G
MNv4-Hybrid-L (46)	7.2 G
Our Complex Net (CIFAR 100)	337 M
UWYN (CIFAR 100)	1186 M

A.7 Feasibility of our early exit

To explore if our concept of early exit is feasible or not, we have implemented the early exit from on a ViT (23). This indicates that after execution of each Transformer block, the output was sent to the classifier, and based on the classifier features and labels of the images, a cross-entropy loss was implemented. Table 9 demonstrates the results when we train these pipelines over a limited number of epochs, thereby testifying the feasibility of our work. We start the early exit after and confidence score calculation after at least 4 transformer blocks during inference.

Table 9: **Possibility of Early Exit:** Performance comparison of full capacity vs. early exit ViTs across various datasets, trained from scratch for 100 epochs. This table illustrates the generalizability of our method, showing that performance acceleration is consistent across different datasets and not solely reliant on the patch and attention head selector networks. From the full capacity accuracy (Acc.), there is a limited dip by 1%, while the other metrics, such as MACs, Parameters (Params), and Inference time (Time), have reduced significantly.

Metric	CIFAR-10 (Full Capacity)	BloodMNIST (Full Capacity)	CIFAR-10 (Early Capacity)	BloodMNIST (Early Exit)
Accuracy (%)	80.21	97.02	80.08	96.89
MACs (M)	1384	1401.8	836.6	596.4
Params (M)	21.31	21.31	12.34	12.34
Time (sec)	273.07	12.52	247.09	5.59

A.8 Other motivational examples

As mentioned in the Introduction, Section 1, the redundancy in transformers is also apparent in Natural Language Processing tasks as well. We examine BART (26), a widely used transformer model comprising 12 encoder and 12 decoder blocks, in the context of text classification task on the MNLI dataset (60). In this task, Lewis *et al*. (26) categorizes if a pair of sentences as contradictory or not. We systematically drop later blocks in both the encoder and decoder to evaluate the impact on performance and computational efficiency. As Table 10 illustrates, reducing the number of blocks yields significant computational savings with only minor accuracy decreases. For instance, using 10 encoder blocks and 8 decoder blocks results in a mere 0.07% accuracy reduction while saving approximately 30 ms per CPU and GPU computation time, reducing FLOPs by almost 25%. These findings suggest a trade-off between accuracy and efficiency, which could be leveraged for applications where rapid inference is critical or resources are constrained.

Table 10: **Motivation from NLP:** Experimental results of BART (26) on the MNLI dataset, demonstrating the impact of reducing model components (encoder/decoder blocks) on computational demand and accuracy.

Encoders	Decoders	Params	CPU (ms)	GPU (ms)	KFLOPs	Accuracy (%)
12	12	4.07×10^8	81.51	87.73	12.03×10^3	83
10	10	3.49×10^8	51.77	61.08	10.02×10^3	82.35
8	8	2.90×10^8	46.52	50.19	8.02×10^3	52.66
10	8	3.15×10^8	53.86	57.76	8.88×10^3	82.93
10	6	2.81×10^8	37.58	34.55	7.73×10^3	70.08

Table 10 highlights trade-offs between efficiency (CPU/GPU time for inference on the entire test set, parameters, KFLOPs) and classification performance, supporting the hypothesis that not all model components are essential for model efficiency. This indicates that intermediate features are also well learnt in most cases.