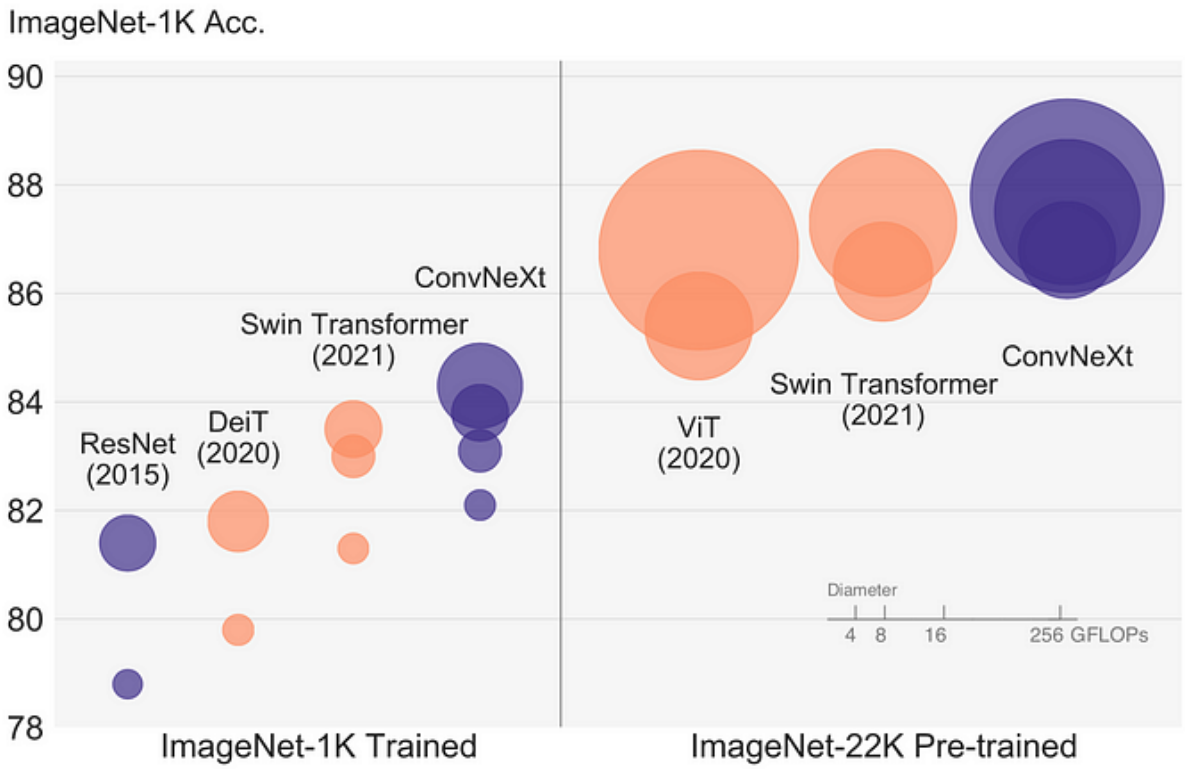
Clothing Article Classification Using ConvNEXT Tiny



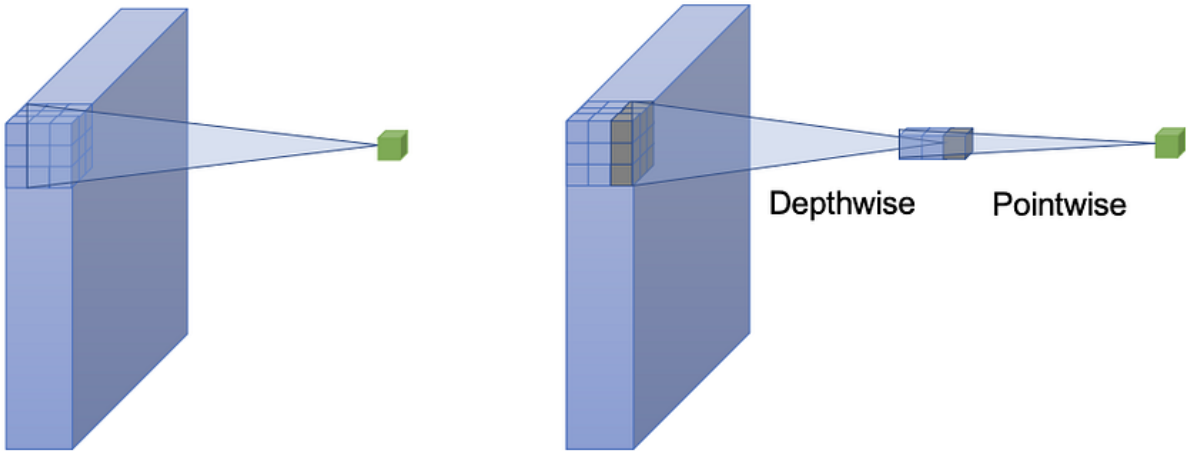
For this Task: We explored four different approaches:

1. Custom CNN without any pre-trained models.
2. ResNet50 with 17 classes.
3. ResNet50 with only the top 10 classes to handle the imbalanced dataset.
4. ConvNEXT-T feature extraction model from TensorFlow Hub.

After comparing the performance and efficiency of these models, we chose the ConvNEXT tiny feature extraction model from TensorFlow Hub as our final approach, as it performed well on the imbalanced dataset. We then applied 5-fold cross-validation to the ConvNEXT tiny feature extraction model to ensure its highest accuracy and robustness.

ConvNext Approach

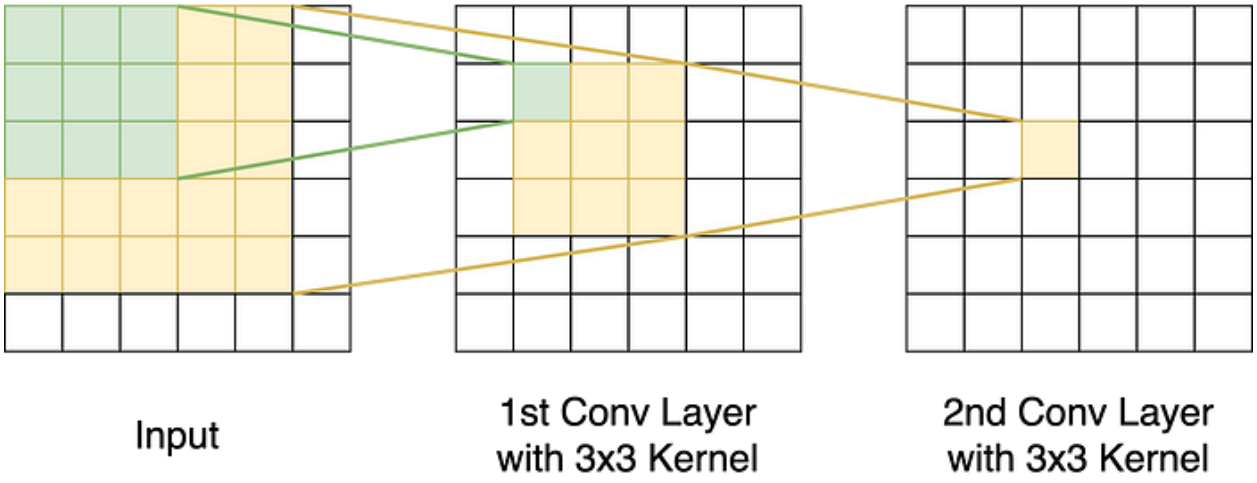
We will focus on parts of the ConvNEXT approach because it is huge. ConvNEXTt authors start with a standard ResNet (e.g. ResNet50) and gradually "*modernize*" the architecture to the construction of a hierarchical vision Transformer (e.g. Swin-T).



Standard Convolution, Depthwise Convolution & Pointwise Convolution

ConvNEXT uses depthwise convolution, a special case of grouped convolution where the number of groups equals the number of channels. Depthwise convolution is similar to the weighted sum operation in self-attention, which operates on a per-channel basis, i.e., only mixing information in the spatial dimension.

Effective Receptive Field in Deep Convolutional Neural Networks



In deep networks, a receptive field — or field of view — is the region in the input space that affects the features of a particular layer as shown in Fig.1. The receptive field is important for understanding and diagnosing a network's performance. A deep networks should be designed with a receptive field that covers the entire relevant image region because the network is oblivious to regions outside its receptive field and because the receptive field determines the spatial extent to which the model can capture information.

Calculating the receptive field:

$$r_0 = \sum_{l=1}^{L} \left((k_l - 1) \prod_{i=1}^{l-1} s_i \right) + 1$$

r_0 → receptive field of our input layer.

L → number of layer.

k_l → kernel size.

s_l → stride.

Adjusting the receptive field:

To adjust the receptive field of any model, we can modify the architecture by changing the kernel sizes, strides, or adding/removing layers.

To Increase the receptive field:

- **Use larger kernel sizes:** Increasing the kernel size of convolutional layers will directly increase the receptive field. For example, changing the kernel size from 3x3 to 5x5 will result in a larger receptive field.

- **Increase the stride:** Increasing the stride of convolutional layers will also increase the receptive field, as the output neurons will have a larger spatial gap between them. For example, changing the stride from 1 to 2 will increase the receptive field.

- **Add more layers:** Adding more convolutional layers, especially with larger kernel sizes or strides, can indirectly increase the overall receptive field of the model.

To Decrease the receptive field:

- **Use smaller kernel sizes:** Decreasing the kernel size of convolutional layers will reduce the receptive field. For example, changing the kernel size from 5x5 to 3x3 will result in a smaller receptive field.

- **Decrease the stride:** Decreasing the stride of convolutional layers will reduce the receptive field, as the output neurons will have a smaller spatial gap between them. For example, changing the stride from 2 to 1 will decrease the receptive field.

- **Remove layers:** Removing convolutional layers, especially those with larger kernel sizes or strides, can indirectly decrease the overall receptive field of the model.

So, adjusting the receptive field may impact the model's performance. A larger receptive field allows the model to capture more global information, which can be beneficial for tasks involving large structures or patterns. On the other hand, a smaller receptive field may be more suitable for tasks that require capturing local information or fine-grained patterns.

Computational Complexity

We calculated the number of Floating-Point Operations (FLOPS) and Multiply-accumulate operations (MACCs) per layer for the main convolutional and fully connected layers in the three models: Custom CNN and ResNet50. The results are presented in the table below the most expensive layers:

Model	Layer Name	FLOPSx 10^6	MACCsx 10^6
Custom CNN	Conv2	9,912.32	4,956.16
ResNet50	ResBlock1_Conv1	248.84	124.42

Total FLOPS and MACCs for each Model:

Model	Total FLOPSx 10^6	Total MACCsx 10^6
Custom CNN	29,891.94	14,945.97
ResNet50	7,735.46	3,867.73
ConvNext-T	4,500.46	2,250.23

Note: The Flops of ConvNext-T couldn't get it from the calculations because the TensorFlow Hub model is closed, but in future i will try to revert it to its architecture or use the PyTorch model. So, the Flops in the table is from its benchmark.

Decreasing FLOPS and MACCs:

To decrease the FLOPS and MACCs, we can apply various techniques, such as:

1. **Reduce the number of filters or neurons:** By reducing the number of filters in convolutional layers or neurons in fully connected layers, we can decrease the number of operations performed, thereby lowering FLOPs and MACCs.

2. **Apply network pruning:** Network pruning involves removing less important connections or weights from the network based on some criteria, such as their magnitude. By pruning the network, we can reduce its complexity without sacrificing much performance.

3. **Reduce input size:** Decreasing the spatial dimensions of the input image can lower the number of operations performed in each layer, reducing FLOPs and MACCs.

4. **Employ model compression techniques:** Techniques like quantization, knowledge distillation, and weight sharing can help reduce FLOPs and MACCs. Quantization reduces the precision of weights and activations, leading to a decrease in computational complexity. Knowledge distillation trains a smaller student network to mimic the behavior of a larger teacher network. Weight sharing involves grouping similar weights together, reducing the number of unique weights and operations.

Applying these techniques may impact the overall performance of the model. It's essential to find a balance between reducing FLOPs and MACCs and maintaining acceptable accuracy and performance for our specific task.

Most Computationally Expensive Layers

I have added a table with the most expensive layer in each model, along with the time it takes to compute.

Model	Most Expensive Layer	Time (ms)
Custom CNN	Batch-Normalization	0.055
ResNet50	ResBlock3_Conv3	0.45
ConvNext-T	ConvNext_Block_FE	0.07

In the table above, the most computationally expensive layers are typically the convolutional layers with the highest number of filters and the largest filter sizes. In the case of ResNet50, the most expensive layers is the ones within the residual blocks.

By focusing on optimizing these layers, we can significantly decrease the overall computational complexity of the models.