Architectural Optimizations

Aaron Zhao, Imperial College London

Introduction

Efficiency is a key metric in evaluating performance

- Re-design the basic operands
- Architecture level re-engineering
- System-level re-structuring

Most of these modifications are at the algorithmic level!

How can we modify the networks to make them more efficient?

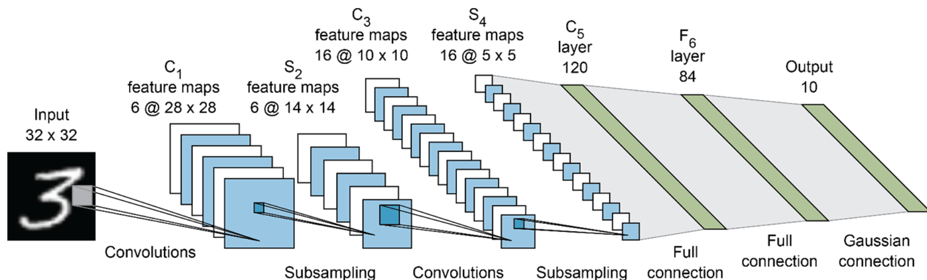
I will go through three pieces of work in detail

- Early versions of MobileNet – Depthwise Separable Convolution
- Longformer – A local windowed attention
- MobileViT – Hybrid Models
- LLaMA – KV Cache

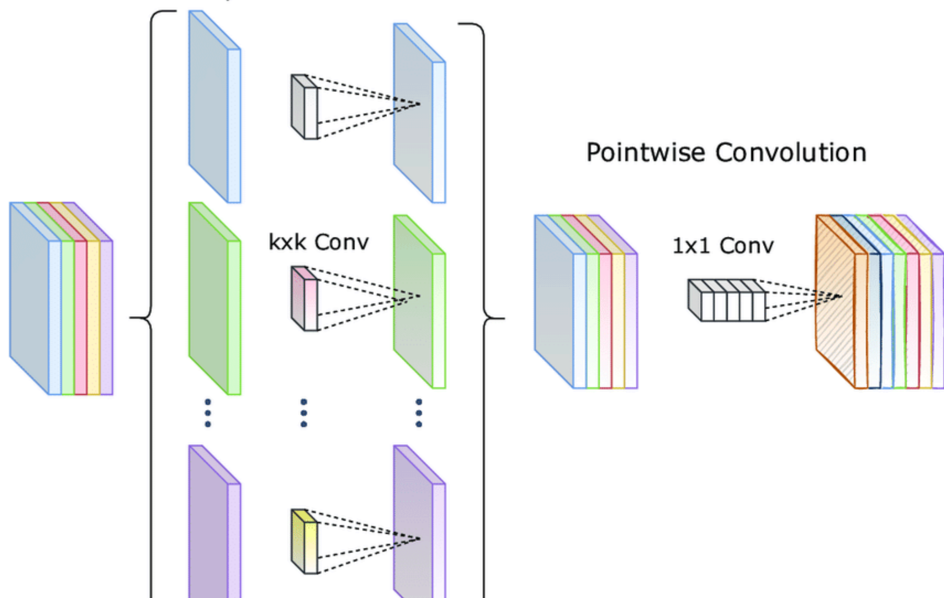
MobileNet

Convolution As we have mentioned before, $(N, C_{in}, C_{out}, K, H, W)$ roughly defines the operation.

Parameters: $C_{in} \times C_{out} \times K \times K$



Depthwise Separable Convolution



Depthwise Separable Convolution (ii)

The core idea is basically decomposition.

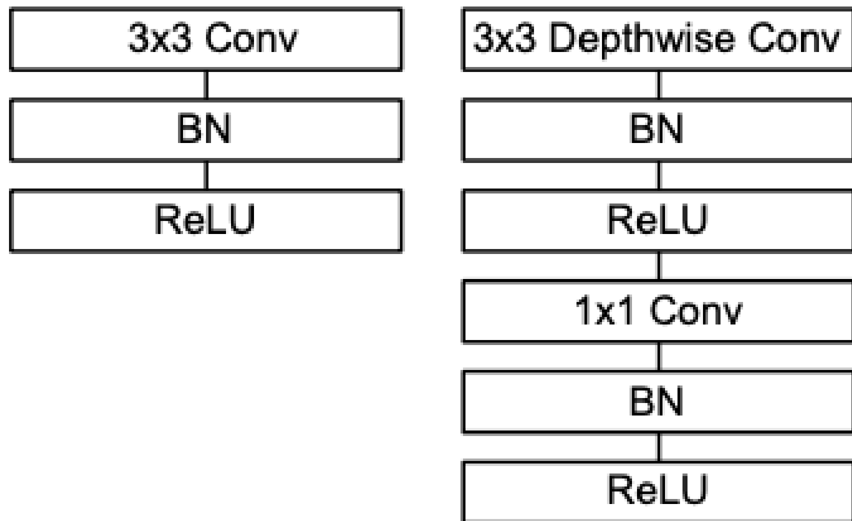
Depthwise Separable Convolution is made of a depthwise convolution and a pointwise convolution.

- Depthwise Convolution: grouped convoltuion, where the group size equals to the number of channels.
- Pointwise Convolution: convolution with a kernel size of 1.

Convolution Parameters: $C_{in} \times C_{out} \times K \times K$

Depthwise Separable Convolution Parameters: $C_{in} \times K \times K + C_{in} \times C_{out}$

Depthwise Separable Convolution



Depthwise Separable Convolution (ii)

Multiple ReLU and BN layers are added to make up the block.

BN: Batch Normalization.

Table 1. MobileNet Body Architecture

Type / Stride		Filter Shape	Input Size
Conv / s2		$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1		$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1		$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2		$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1		$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1		$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1		$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2		$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1		$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1		$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1		$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2		$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1		$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$	Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
	Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2		$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1		$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2		$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1		$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1		Pool 7×7	$7 \times 7 \times 1024$
FC / s1		1024×1000	$1 \times 1 \times 1024$
Softmax / s1		Classifier	$1 \times 1 \times 1000$

You should be comfortable with reading an architecture table like this.

Longformer

The context length problem

When dealing with a Transformer model, we face the N^2 curse from the full attention computation, where N is the sequence length.

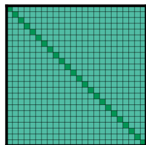
Recall that

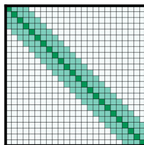
$$Atten(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

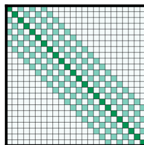
If the context length is large, this means the sequence length N is large.

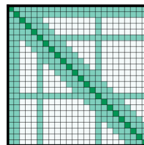
We have the operation QK^T and $Q, K \in \mathcal{R}^{N \times d}$, the complexity of this operation is then $O(N^2)$.

Longformer


(a) Full n^2 attention

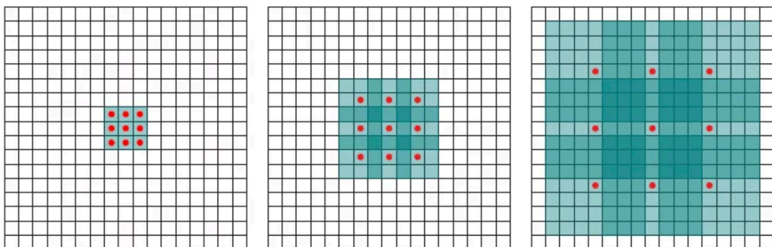

(b) Sliding window attention


(c) Dilated sliding window


(d) Global+sliding window

- Use dilated sliding window attention to compute a small number of diagonals.
- Global attention on pre-selected fixed entries (based on certain heuristics).
- Requires a CUDA implementation to get a true speedup.

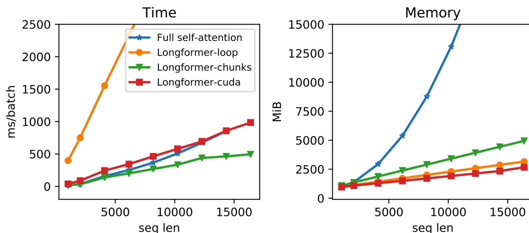
What is dilation?



l=1 (left), l=2 (Middle), l=4 (Right)

- An operation that was firstly used in convolutions (dilated convolutions).
- We skip certain middle points in the computation.
- Images are showing dilation factor (i) equals to 1, 2 and 4. Larger dilations have a larger receptive field.

Longformer Performance



- Real performance confirms with the theory: $O(N)$ scaling in memory.
- Different implementation may introduce different time. CUDA kernel is implemented using TVM, it might be faster if native CUDA is used.

Re-design the basic operands

- Philosophy: use cheaper operators to approximate the standard operators
- Rely on SGD training from scratch to empirically verify performance.

MobileViT

The ViT structure

Vision Transformer (ViT)

Class
Bird
Ball
Car
...

MLP
Head

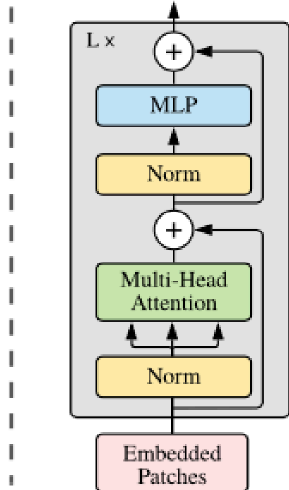
Transformer Encoder

Patch + Position
Embedding

* Extra learnable
[class] embedding

0 * 1 2 3 4 5 6 7 8 9

Linear Projection of Flattened Patches



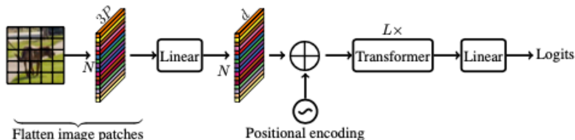
The ViT structure (ii)

MobileViT ViTs are more computationally demanding than CNNs.

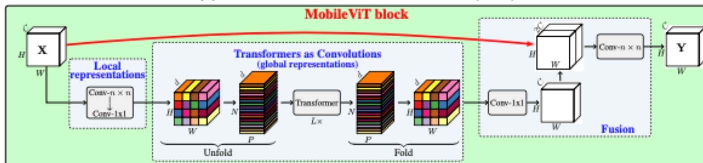
- ViT is more heavy-weight. ViT-B/16 vs. MobileNetv3: 86 vs. 7.5 million parameters.
- More performing (higher accuracy) at a high parameter count does not necessarily mean it is performant at low parameter count.
  - For a parameter budget of about 5-6 million, DeIT is 3% less accurate than MobileNetv3.
- We have more energy-efficient neural operators in CNNs (eg. Depthwise Seperable Convolution!).

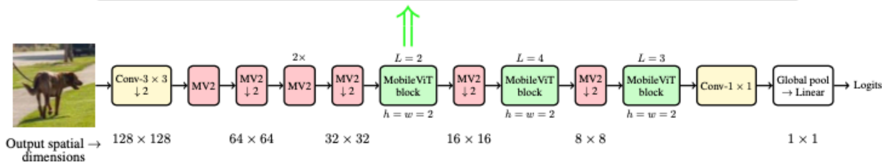
The proposed solution: mix ViT layers with convolutions

MobileViT



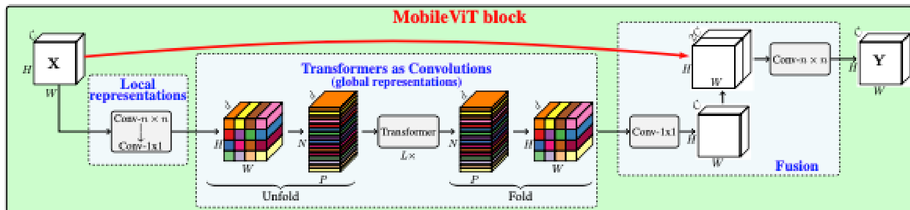
(a) **Standard visual transformer (ViT)**





Add ViT operation after convolution blocks.

MobileViT

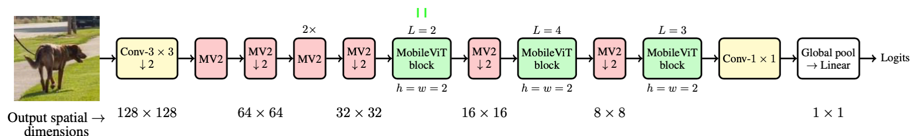


The original value has $\boldsymbol{X} \in \mathcal{R}^{H \times W \times C}$

- Transform $\boldsymbol{X} \in \mathcal{R}^{H \times W \times C}$ to patches $\boldsymbol{X} \in \mathcal{R}^{H \times W \times d}$ using convolutions.
- Unfold to $\boldsymbol{X}_U \in \mathcal{R}^{N \times P \times d}$ and pass through the Transformer blocks.
- Fold back to $\boldsymbol{X}_F \in \mathcal{R}^{H \times W \times d}$.
- Another convolution block to push back to $\boldsymbol{X}_C \in \mathcal{R}^{H \times W \times C}$.

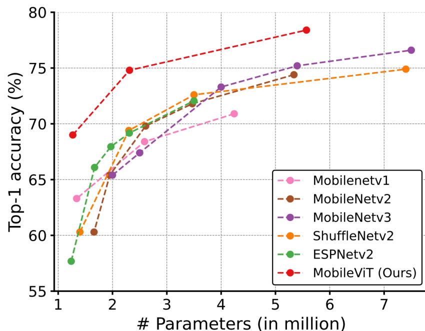
MobileViT final fusion process

- Concatenate $\boldsymbol{X} \in \mathcal{R}^{H \times W \times C}$ and $\boldsymbol{X}_C \in \mathcal{R}^{H \times W \times C}$.
- $[\boldsymbol{X}, \boldsymbol{X}_C] \rightarrow \boldsymbol{X}_{out}$
- $\boldsymbol{X}_{out} \in \mathcal{R}^{H \times W \times C}$



- Fusion of the two neural operators (convolution and vit).
- Parameter-efficient convolutions for local information.
- ViT structure for global information.

Architecture level re-engineering



- Pick and match different operators in an informed way.
- We will look at more of this style of optimization later (Network Architecture Search).
- Use the Pareto Frontier to judge whether you are doing better!

KV Caching

Decoder-only transformers

If you actually think about how is an output sentence generated, you might find that this is an iterative process:

```
i = 0
while out_token != token_eos:
  logits, _ = model(in_tokens)
  out_token = torch.argmax(
    logits[-1, :], dim=0, keepdim=True),
    in_tokens=torch.cat((in_tokens, out_token), 0)
  text = tokenizer.decode(in_tokens)
  print(f'step i input: text', flush=True)
  i += 1
```

Decoder-only transformers

- For each input (question sentence), we generate a single token.
- We then append this output token to the input token.
- We use the appended sequence to run inference again until we see an 'EOS' token.

```
# step 0
Lionel Messi is a player
# step 1
Lionel Messi is a player who
# step 2
Lionel Messi is a player who has ...

Input: Lionel Messi is a
Output: Lionel Messi is a player who has been a key part of the
team's success.
```

This is very expensive, the compute cost scales quadratically with sequence length!

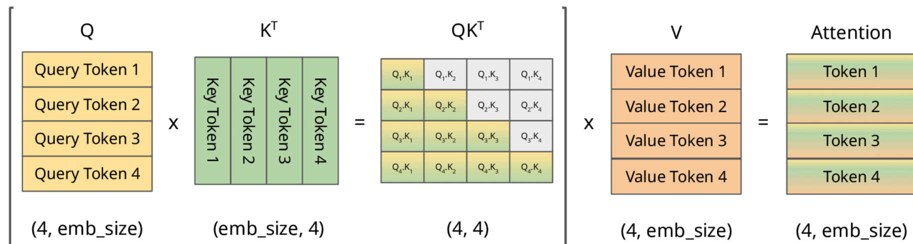
The idea of Caching

However, if we think carefully, when computing the ith token, we have already generated the previous intermediate values for all previous (0 to $i - 1$) tokens.

Since the decoder is causal (i.e., the attention of a token only depends on its preceding tokens), at each generation step we are recalculating the same previous token attention, when we actually just want to calculate the attention for the new token.

Note, this is CLM (Casual Language Modeling), where we have a CLM mask to mask out the upper parts in QK^T.

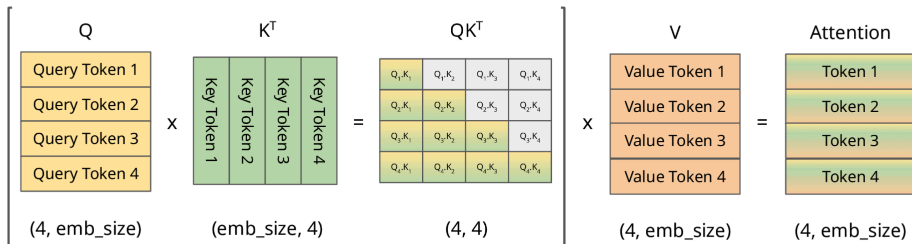
The idea of Caching (ii)



Classic compute

1. Compute Q_1K_1
2. Compute Q_1K_1, Q_2K_1, Q_2K_2
3. Compute $Q_1K_1, Q_2K_1, Q_2K_2, Q_3K_1, Q_3K_2, Q_3K_3$

The idea of Caching (iii)



Compute with the KV cache

1. Compute $Q_1 K_1$, put this into cache
2. Take $Q_1 K_1$ from cache, compute $Q_2 K_1, Q_2 K_2$, add $Q_2 K_1, Q_2 K_2$ into cache
3. Take $Q_1 K_1, Q_2 K_1, Q_2 K_2$ from cache, compute $Q_3 K_1, Q_3 K_2, Q_3 K_3$, add $Q_3 K_1, Q_3 K_2, Q_3 K_3$ into cache

Summary

- Re-design the basic operands
  - Depthwise Separable Convolutions
  - Longformer
- Architecture-level re-engineering
  - MobileVit
- System-level re-structuring
  - KV Caching in LLaMA