Week 14

Master Thesis 2020

Tobias Engelhardt Rasmussen (s153057)

DTU Compute

December 17, 2020

Outline

Since Last

- FLOPs (Floating Point Operations - + - * %) for calculating theoretical speed-up
- Timing - Found a timing function that seems to give stable results
- Writing - Methodology and appendices

Outline

FLOPs given a network layer

Stolen:

A.1 FLOPs COMPUTATION

To compute the number of floating-point operations (FLOPs), we assume convolution is implemented as a sliding window and that the nonlinearity function is computed for free. For convolutional kernels we have:

$$\text{FLOPs} = 2HW(C_{in}K^2 + 1)C_{out}, \tag{11}$$

where H, W and C_{in} are height, width and number of channels of the input feature map, K is the kernel width (assumed to be symmetric), and C_{out} is the number of output channels.

For fully connected layers we compute FLOPs as:

$$\text{FLOPs} = (2I - 1)O, \tag{12}$$

where I is the input dimensionality and O is the output dimensionality.

Convolutional FLOPs maybe a bit too simple... Does not take padding or stride into account.
Made my own calculations based on the size of the output tensor with very similar results...

FLOPs given a network

My calculations for convolutions based on the number of output values:

$$FLOPs = \underbrace{T \cdot F' \cdot H' \cdot W' \cdot}_{\text{Size of output}} \left(2 \cdot \underbrace{S \cdot K^3}_{\text{Size of filter}} - 1 \right)$$

Which would correspond to: (for 2D)

$$FLOPs = C_{out} \cdot H' \cdot W' \cdot \left(2 \cdot C_{in} K^2 - 1 \right)$$

While theirs is (rearranged):

$$FLOPs = C_{out} \cdot H \cdot W \cdot 2 \cdot \left(C_{in} K^2 + 1 \right)$$

Outline

Since last

FLOPs for theoretical speed-up

Timing

Writing

Timing results

Number of parameters	3D conv 1	3D conv 2	Linear 1	Linear 2	Linear 3	**Total**
Original	14,520	58,080	358,400	10,752	168	441,920
Compressed	4,852	3,690	14,266	212	168	23,188
Ratio	0.334	0.064	0.039	0.020	1	0.052

The timing is performed on a CPU

	Timing	3D conv 1	3D conv 2	Linear 1	Linear 2	Linear 3	**Total**
FLOPs (1000)	Original	15,618,892	697,158	716.8	21.5	0.34	16,316,789.4
	Compressed	1,943,628	27,490.1	28.53	0.4	0.34	1,971,147.4
	Speed-up	8.0359	25.3603	25.1288	50.8369	1	**8.2778**
Time (s)	Original	2.6807	0.01623	$1.3 \cdot 10^{-5}$	$5.6 \cdot 10^{-7}$	$1.1 \cdot 10^{-7}$	2.9594
	Compressed	1.1688	0.007246	$2.1 \cdot 10^{-8}$	$2.1 \cdot 10^{-8}$	$9.8 \cdot 10^{-8}$	1.0805
	Speed-up	2.2936	2.2434	6.4310	2.5622	1.1249	**2.7390**
Accounts for approx. (%)		95.7	4.27	$4.4 \cdot 10^{-3}$	$1 \cdot 10^{-4}$	$2 \cdot 10^{-6}$	100

Outline

Methodology Section

Overall outline:

- Decomposing the Input
  - Estimating the loadings for the testing data
- Compressing a pre-trained Network Using Tucker
  - The linear / dense layer
  - The convolutional layer
  - Rank selection
  - One-shot Compression of an entire CNN using Tucker

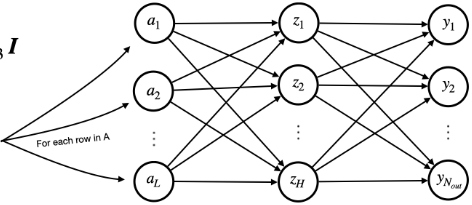
Decomposing the Input

For MNIST:

$$\mathcal{X}^{N\times28\times28} \approx$$

$$\mathcal{G}^{L\times28\times28} \times_1 \underbrace{\boldsymbol{A}^{N\times L}} \times_2 \boldsymbol{I} \times_3 \boldsymbol{I}$$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1L} \\ a_{21} & a_{22} & \cdots & a_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NL} \end{bmatrix}$$



One-Shot Compression of an Entire CNN Using Tucker

Algorithm 1 One-Shot Tucker Compression of CNN

`net` ← define an appropriate CNN
train `net` using the given training data
`net_dcmp` ← make a copy of `net`
for each **convolutional** layer; `layer` in `net_dcmp` **do**
  $\mathcal{K}_{\texttt{layer}}$ ← take out weight tensor
  R_4, R_5 ← choose appropriate rank(s)
  $\mathcal{G}$, $\boldsymbol{U}^{(4)}, \boldsymbol{U}^{(5)}$ ← Decompose $\mathcal{K}_{\texttt{layer}}$ using relevant algorithm ▷ Or just one $\boldsymbol{U}$
  `layer_dcmp_1` ← define new 1×1 convolution ▷ If applicable
  `layer_dcmp_2` ← define new 3D convolution
  `layer_dcmp_3` ← define new 1×1 convolution ▷ If applicable
  $\mathcal{K}_{\texttt{layer\_dcmp\_1}} \leftarrow \boldsymbol{U}^{(4)}$ ▷ If applicable
  $\mathcal{K}_{\texttt{layer\_dcmp\_2}} \leftarrow \mathcal{G}$
  $\mathcal{K}_{\texttt{layer\_dcmp\_3}} \leftarrow \boldsymbol{U}^{(5)}$ ▷ If applicable
  $\boldsymbol{b}_{\texttt{layer\_dcmp\_3}} \leftarrow \boldsymbol{b}_{\texttt{layer}}$ add the bias to the last layer ▷ Or in the line above
  `layer` ← `sequence(layer_dcmp_1, layer_dcmp_2, layer_dcmp_3)`
end for
for each **linear** layer; `layer` in `net_dcmp` **do**
  $\boldsymbol{W}_{\texttt{layer}}$ ← take out weight matrix of size $N_{out} \times N_{in}$
  R_A, R_B ← choose appropriate rank(s)
  $\boldsymbol{G}, \boldsymbol{A}, \boldsymbol{B}$ ← decompose $\boldsymbol{W}_{\texttt{layer}}$ using relevant algorithm ▷ Or just $\boldsymbol{A}$ or $\boldsymbol{B}$
  `layer_dcmp_1` ← define new $R_B \times N_{in}$ linear layer ▷ If applicable
  `layer_dcmp_2` ← define new $R_A \times R_B$ linear layer ▷ Or $R_B \times N_{in}$ or $N_{out} \times R_A$
  `layer_dcmp_3` ← define new $N_{out} \times R_A$ linear layer ▷ If applicable
  $\boldsymbol{W}_{\texttt{layer\_dcmp\_1}} \leftarrow \boldsymbol{B}$ ▷ If applicable
  $\boldsymbol{W}_{\texttt{layer\_dcmp\_2}} \leftarrow \boldsymbol{G}$
  $\boldsymbol{W}_{\texttt{layer\_dcmp\_3}} \leftarrow \boldsymbol{A}$ ▷ If applicable
  $\boldsymbol{b}_{\texttt{layer\_dcmp\_3}} \leftarrow \boldsymbol{b}_{\texttt{layer}}$ add the bias to the last layer ▷ Or in the line above
  `layer` ← `sequence(layer_dcmp_1, layer_dcmp_2, layer_dcmp_3)`
end for
train `net_dcmp` using the given training data ▷ fine-tuning