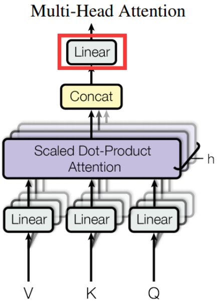
A Workflow for High Performance and High Fidelity Neural Network Compression

Xuanhang Diao, Letong Han, Yifan Chen

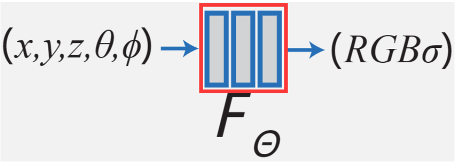
May 6, 2023

Idea

Multi-layer perceptron(MLP) is a simple but effective structure, which is quite common in various neuron network designs.



(a) Transformer



(b) NeRF

MLP

The large number of parameters limits its inference speed.

- $$FLOPs_{CNN} = 2 \times H \times W \times C_{in} \times K^2 \times C_{out}$$
- $$FLOPs_{MLP} = N_{in} \times N_{out} + N_{out} \sim (H \times W)^2 \times C_{in} \times C_{out}$$

We have to think about reducing the model size and speed up inference.

Neural network compression

Common neural network compression methods

- Pruning
- Quantization
- Tensor Decomposition

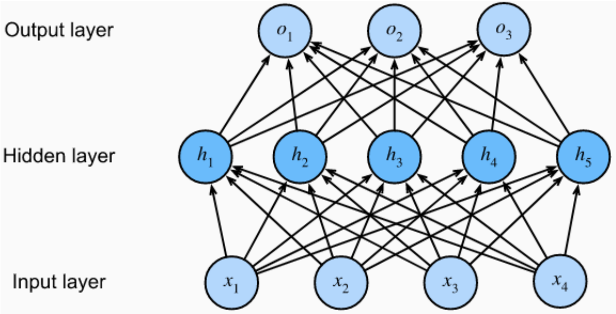


Figure: An MLP with a hidden layer of 5 hidden units

Neural network compression

Model	Inference time(s)	Model size(KB)	PSNR
original	0.597	2382.301	27.14
INT8	0.241	616.341	21.77

- Improved Inference speed(2.47$\times$)
- Decline in model size(3.87$\times$)
- Image quality is reduced

Insufficient and motivation

Troubles

- Simple INT8 quantization is not enough
- It will take more time on a larger MLP (remember the size of the fully connected layer is related to the square of the input and output)

Target

MLP compression with

- High speed
- Low precision loss

Outline

We consider a two stage model compression/acceleration strategy

1. Tensor Decomposition
2. Quantization

Describe with the following order

1. Principles
2. Combine with our questions
3. Parallel acceleration

Low-rank representation of fully connected layers

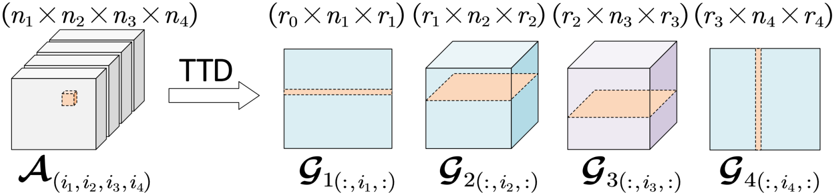
Fully-connected layers apply a linear transformation to an N-dimensional input vector x

$$y = Wx + b$$

A TT-layer transforms a d-dimensional tensor χ (formed from the corresponding vector x) to the d-dimensional tensor Γ (which correspond to the output vector y). We assume that the weight matrix W is represented in the TT-format with the cores $G_k[i_k, j_k]$. Finally the linear transformation of a fully-connected layer can be expressed in the tensor form

$$\gamma(i_1, ..., i_d) \sum_{j_1,...,j_d} G_1[i_1, j_1]...G_d[i_d, j_d]\chi(j_1, ..., j_d) + \beta(i_1, ..., i_d)$$

Tensor Train (TT)-format



The representation of a tensor A via the explicit enumeration of all its elements requires to store $\Pi_{k=1}^{d} n_k$ numbers compared with $\sum_{k=1}^{d} n_k r_{k-1} r_k$ numbers if the tensor is stored in the TT-format. Thus, the TT-format is very efficient in terms of memory if the ranks are small[1].

---

[1] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. In Advances in Neural Information Processing Systems, pages 442–450, 2015.

Formal Definition of approximation Problem

The weight matrix W can be decomposed into the product of multiple Tensors

$$W_i = Q_i^1 \cdot Q_i^2$$

where $Q_i^1 \in \mathbb{R}^{1\times n_1 \times r}$ and $Q_i^2 \in \mathbb{R}^{r\times n_2 \times 1}$ are the TT cores, and r is the target rank controlling the compression ratio.

- Straightforwardly decomposing the full-rank tensor W_i into a low-rank TT format inevitably causes a large approximation error,

we adopt this ADMM-based low rank approximation to finetune the learned NeRF.

$$\begin{aligned} & argmin_{\{W_i, b_i\}} ||c' - c_{gt}||_2^2 \\ & s.t. \quad rank(W_i) < r, \end{aligned}$$

where r is the desired rank of W_i.

Why ADMM

The alternating direction method of multipliers (ADMM) is an algorithm that solves convex optimization problems by breaking them into smaller pieces, each of which are then easier to handle.

- Can be decomposed into multiple unrelated sub-problems to solve to achieve parallelism
- Converge in most practical problems

basic ADMM

Most of the problems that meet the following conditions can be optimized using ADMM (without concern of convergence issues)

- Two optimization variables
- Only equality constraints

Consider a problem of the form

$$\min f(x) + g(z)$$
$$s.t.\ Ax + Bz = c$$

basic ADMM (cont.)

Turn Primal problem to Lagrangian function

$$L_\rho(x, z, u) = f(x) + g(z) + u^T(Ax + Bz - c)$$

define augmented Lagrangian, for a parameter $\rho > 0$

$$L_\rho(x, z, u) = f(x) + g(z) + u^T(Ax + Bz - c) + \frac{\rho}{2}\|Ax + Bz - c\|_2^2$$

basic ADMM (cont.)

Repeat for $k = 1, 2, 3, \ldots$

$$
\begin{aligned}
x^{(k)} &= argmin_x L_\rho(x, z^{(k-1)}, u^{(k-1)}) \\
z^{(k)} &= argmin_z L_\rho(x^{(k)}, z, u^{(k-1)}) \\
u^{(k)} &= u^{(k-1)} + \rho(Ax^{(k)} + Bz^{(k)} - c)
\end{aligned}
$$

- Transform a multi-objective problem into multiple single-objective optimization problems
- For single-objective optimization, refer to common single-objective optimization algorithms such as gradient descent (GD)

Distributed ADMM

needs to compensate A and c as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \dots & \mathbf{A}_{1N} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \dots & \mathbf{A}_{2N} \\ & & \dots & \\ \mathbf{A}_{M1} & \mathbf{A}_{M2} & \dots & \mathbf{A}_{MN} \end{bmatrix}, \mathbf{b} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \dots \\ \mathbf{b}_M \end{bmatrix}$$

Distributed ADMM(cont.)

The problem transfer to

$$
\begin{aligned}
\min \sum_j f_j(x_j) + \sum_i g_i(z_i) \\
s.t. \ \sum_j P_{i,j} x_j + z_i = b_i, \forall i \\
P_{i,j} = A_{i,j} X_j, \forall i, j \\
x_j = x_{ij}, \forall i, j
\end{aligned}
$$

Solved the optimization problem in ADMM way

This non-convex optimization with constraints can be solved using the ADMM optimization method by introducing an auxiliary variable Z and an indicator function $g(\cdot)$

$$g(W_i) = \begin{cases} 0, rank(W_i) < r \\ +\infty, otherwise. \end{cases}$$

The original optimization problem can then be rewritten

$$\begin{aligned} & argmin_{\{W_i, Z_i\}} ||c' - c_{gt}||_2^2 + g(Z_i) \\ & s.t. \quad W_i = Z_i \end{aligned}$$

Solved the optimization problem in ADMM way(cont.)

Refer to the general steps of ADMM

- Augmented Lagrangian

$$\mathcal{L}(W_i, Z_i, U_i) = l(W_i) + g(Z_i) + \frac{\rho}{2}||W_i - Z_i + U_i||_F^2 + \frac{\rho}{2}||U_i||_F^2,$$

- Iterative solution of multiple single-objective optimization problems

$$\begin{aligned} W_i^{t+1} &= argmin_{\{W_i\}}\mathcal{L}(W_i^t, Z_i^t, U_i^t), \\ Z_i^{t+1} &= argmin_{\{Z_i\}}\mathcal{L}(W_i^{t+1}, Z_i^t, U_i^t), \\ U_i^{t+1} &= U_i^t + W_i^{t+1} - Z_i^{t+1} \end{aligned}$$

Solved the optimization problem in ADMM way(cont.)

- W-subproblem

$$\min_{W} \quad l(W_i) + \frac{\rho}{2}||W_i - Z_i + U_i||$$

Can be solved by Stochastic Gradient Descent (SGD)

- Z-subproblem

$$\min_{Z} \quad g(Z_i) + \frac{\rho}{2}||W_i - Z_i + U_i||$$

Recall that

$$g(Z_i) = \begin{cases} 0, rank(Z_i) < r \\ +\infty, otherwise. \end{cases}$$

and $g(\cdot)$ is not differentiable

Solved the optimization problem in ADMM way(cont.)

- Z-subproblem
  According to [2], updating Z can be performed as:

$$Z_{i+1} = \Pi_S(W_{i+1} + U_t)$$

  Where $\Pi_S(\cdot)$ is the projection of singular values onto S, which is done by truncating ranks to target ranks r^*.
- Singular Value Decomposition (SVD) for real matrices

$$M = U\Sigma V^T$$

  M:$m \times n$ matrix
  U:$m \times m$ orthogonal unitary matrix, Σ:$m \times n$ diagonal matrix,
  V:$n \times n$ orthogonal unitary matrix,

[2] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. 2011. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. Found. Trends Mach. Learn. 3, 1 (January 2011), 1–122.

Basic idea of DNN Quantization

Weights in deep neural networks can be "compressed"

- similar weights can be replaced by the same value
- representation of weights can use lower precision (like $float32 \rightarrow int8$)

In actual tests, weights usually follow the following rules

- $\mathbf{W} \sim N(0, \sigma^2)$
- The larger the absolute value of the weight, the more "important"

Quantization Methods

So our neural network quantization can follow the following rules

- Biases and the weights outside $[-1, 1]$ are quantized using a uniform 16 bits quantizer to keep enough precision
- the weights within $[-1, 1]$ are quantized

Quantization Methods(cont.)

- Weight Sharing guided by K-means
  codebook is obtained by applying the k-means method to cluster weights into N centroids($N = 256$).
  learn a unique codebook C to quantize the weights within [-1,1] in all layers
- Huffman Coding
  further reduce the network storage by 20% $\sim$ 30%.

Parallel K-means

Consider N data points with P computing units.

1. Assign N/P data points to each unit,
2. Randomly choose K points and assigns them as cluster means and broadcast,
3. In each unit for each data point find membership using the cluster mean,
4. Recalculate local means for each cluster in each processor,
5. Globally broadcast all local means for each processor find the global mean,
6. Go to step (3) and repeat until the terminate condition(eg. iterations > 10000 or number of points where membership has changed is less than 0.1%)

Parallel Huffman Decoding

- According to paper *Revisiting Huffman Coding: Toward Extreme Performance on Modern GPU Architectures*
- Nvidia CUDA toolkit nvJPEG

Future

- Our method is not limited to the only application scenario of NeRF. And all neural networks can perform compression based inference acceleration.
- Our approach is not limited to MLP, but can also be extended to other structures such as CNN and Transformer.

The End

Thank you!