ECE 587 – Hardware/Software Co-Design
Lecture 20 Quantization

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

March 31, 2025

Outline

Reading Assignment

- This lecture: Quantization
  - Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference, Jacob et al. `https://arxiv.org/abs/1712.05877`
- Next lecture: Large Language Models

Outline

Neural Network Model Overview

- A neural network model consists of many layers.
- Inference: generate output from input data, usually
  - A layer uses activations (output) from the previous layer as its input, and computes its activations for the next layer.
  - Within the layer, the input is multiplied with a weight matrix, then a bias vector is added, and finally the result is passed through nonlinear functions like pooling and activation to obtain outputs.
- Training: learn parameters from existing data.
  - Parameters include weight matrices, bias's, etc.
  - Via backpropagation, involving mostly matrix multiplications.
- Parameter sizes matter for both storage and computation.

Growing Complexity

- AlexNet (2012): 60M (million) parameteres
- VGG Net (2014): around 140M parameters
- GPT (Generative Pre-trained Transformer)
  - GPT-1 (2018): 117M parameteres
  - GPT-2 (2019): 1.5B (billion) parameters
  - GPT-3 (2020): up to 175B parameters
- Llama (Large Language Model Meta AI)
  - Llama and Llama 2 (2023): up to 70B parameters
  - Various Llama 3 versions (2024): up to 405B parameters
- DeepSeek V3 (2024) and R1 (2025): 671B parameters
- Is it possible to deploy these models for inference only?
  - To edge devices with limited computational power, memory, storage, and power availability?
  - Trade-off accuracy for latency and cost in general.

Quantization

- Usually, parameters and activations are represented by 32-bit floating point numbers during training.
- Inference with lower bit-depth representations.
  - A 8-bit representation leads to a saving of 4X in storage of weight parameters, and 4X savings in memory for parameters and activations.
  - Representations can be specifically designed to use adders only, eliminating the need of multipliers.
- Challenges: efficiency on commodity hardware without substantial accuracy degradation.
  - Multipliers should be used when they are available.

Quantization Scheme

- Use 8-bit and 32-bit integers for parameters and activations.
  - 8-bit for weights and activations.
  - 32-bit for bias vectors.
- Compute with integer-only arithmatic operations.
  - Without the need to make any conversion or table lookup.
- Uniform quantization
  - There exists an affine mapping between the floating point representation r and the integer representation q.

$$r = S(q - Z)$$

Quantization Parameters

$$r = S(q - Z)$$

- S and Z are quantization parameters.
  - Z: same type as q, representing floating point 0.
  - S: same type as r, representing scale to convert from q to r.
- Use a single pair of (S, Z) for a set of values
  - A weight matrix, a bias vector, an activation vector, etc.
  - Different matrices or vectors may use different pairs of (S, Z)'s.

Scalar Multiplications

$$r_1 = S_1(q_1 - Z_1), r_2 = S_2(q_2 - Z_2), r_3 = S_3(q_3 - Z_3)$$

- If $r_3 = r_1 r_2$, how to compute q_3 from q_2 and q_1?
  - Directly without the need to compute r_1 and r_2 first.
- From $S_3(q_3 - Z_3) = r_3 = r_1 r_2 = S_1 S_2 (q_1 - Z_1)(q_2 - Z_2)$,
  - $q_3 = Z_3 + M(q_1 - Z_1)(q_2 - Z_2)$ where $M = \frac{S_1 S_2}{S_3}$
  - M can be computed offline as $2^{-n} M_0$ where M_0 is a fixed point number in $[0.5, 1)$.
- All operations are integer ones!

Matrix Multiplications

$$r_3^{(i,k)} = \sum_{j=1}^{N} r_1^{(i,j)} r_2^{(j,k)}$$

$$S_3(q_3^{(i,k)} - Z_3) = \sum_{j=1}^{N} S_1 S_2 (q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2)$$

- Let $r_1^{(i,j)}$, $r_2^{(j,k)}$, $r_3^{(i,k)}$ be elements from the three matrices.
- All elements for one matrix share the same S and Z.
- $q_3^{(i,k)} = Z_3 + M \sum_{j=1}^{N} (q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2)$
  - $M = \frac{S_1 S_2}{S_3}$ can be computed offline as the previous slide.
- All operations are integer ones!

Implemention Details

$$q_3^{(i,k)} = Z_3 + M \sum_{j=1}^{N} (q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2)$$

- To obtain accurate results, $q_1^{(i,j)} - Z_1$ and $q_2^{(j,k)} - Z_2$ may require one more bit than that of q's – not convenient.
- For neural network layers, bias is added as $\frac{S_{bias}}{S_3}(q_{bias} - Z_{bias})$
  - Since bias vectors are 32-bit, choose $Z_{bias} = 0$ and $S_{bias} = S_1 S_2$ for simplicity without losing accuracy.
- Expand the above equation to have,

$$q_3^{(i,k)} = Z_3 + M(NZ_1Z_2 + q_{bias} + \sum_{j=1}^{N} q_1^{(i,j)} q_2^{(j,k)} \\ -Z_1 \sum_{j=1}^{N} q_2^{(j,k)} - Z_2 \sum_{j=1}^{N} q_1^{(i,j)})$$

  - $\sum_{j=1}^{N} q_1^{(i,j)} q_2^{(j,k)}$ can be computed as matrix multiplication with 8-bit multipliers and 32-bit accumulators.
  - The rest are computed with 32-bit multipliers and accumulators – not a concern for efficiency since there will be far more operations in the above matrix multiplication.
  - Down-scale the result to 8-bit for $q_3^{(i,k)}$ with saturation.

Post-Training Quantization

- How to obtain quantization parameters and quantized weight matrices and bias vectors?
- Post-training quantization
  - Complete training in floating point numbers
  - For each weight matrix or activation vector, choose a good pair of (S, Z), typically from the range of the values.
- Empirically, post-training quantization works well for large models because there is certain level of redundancy.
- Small models may see significant accuracy drop due to outliers that effective narrow down the range.

Training with Simulated Quantization

- Learn quantization parameters in training.
- Instead of learning (S, Z) directly, learn two parameters (a, b) for each weight matrix and activation vector.
  - As mentioned before, (S, Z)'s for bias vectors are derived from those of weights and activations instead of being learnt.
- During the training process, clamp the values into range $[a, b]$ and then quantize them according to the bit width.
- Intuitively, simulated quantization introduces noise into the training process that the model learns to compensate.
  - So the model will continue to work accurately for inference when quantization introduces the same kind of noise.

Outline

Natural Language Processing (NLP)

- Use natural language as interface between computers and human beings.
- Applications
  - Voice command
  - Machine translation
  - Text summarization
  - Image and video captioning
  - Question answering
  - Story, image, and video generation
  - Many more to come
- Turing test: what is intelligence?

Tokenization

- Convert texts in natural language into tokens that may have meanings to facilitate further processing.
- Character-based tokenization
  - Simple and effective to digitalize texts, e.g. ASCII and Unicode
  - Need extra effort when characters don't carry meanings by themselves, e.g. English.
- Word-based tokenization
  - Encode individual words and punctuations using a vocabulary.
  - How to handle out-of-vocabulary and misspelled words?
  - A very difficult task by itself for languages without word separators, e.g. Chinese.
- Subword tokenization
  - Learn common patterns from character sequences as subword that usually carry meanings and fall back to characters.
  - Handle rare, new, or misspelled words by breaking them into known subword (and characters).

Embedding

- If there are M different tokens, a token can be represented as a $M \times 1$ vector via one-hot encoding.
  - One element is 1 while the rest are 0.
- However, one-hot encoding doesn't capture any meanings.
- Embedding: represent tokens as vectors (usually shorter) to capture semantic relationships and similarities.
  - Tokens are then points in the embedding space.
  - Tokens with similar meanings like 'I' and 'me' are mapped to points that are close in a subspace.
- Assume each vector is of the size $d \times 1$, embedding is learnt during the training process as a $d \times M$ matrix.
- For now on, we will not distinguish between the token and its vector after embedding.

Encoder-Decoder Models

- Most NLP tasks can be formulated as to generate an output sequence of tokens from an input sequence of tokens.
- Since both input and output sequences can have arbitrary lengths, two models are introduced for the NLP task.
  - Encoder $C' = E(C, x)$: process the input sequence of arbitrary length by consuming one token x at a time and transforming the context vector C of fixed size into the next one C'.
  - Decoder $(x, C') = D(C)$: generate the output sequence one token at a time by computing a token x from the context vector C and transforming C into the next one C'.
  - Intuitively, both encoder E and decoder D are FSMs.

Autoregression

- Decoder needs to be statistical: $(Pr, C') = D(C)$
  - Have to learn from natural languages, which are ambiguous and have a lot of variability.
  - Instead of the actual token x, decoder computes Pr as the vector of the probability of each token to be the output.
  - A sampling process then samples Pr to obtain x.
  - But then C' has no knowledge of x – how could the decoder ensure the whole output sequence to be coherent?
- Autoregression: $(Pr, C') = D(C, x^{-1}, x^{-2}, \ldots, x^{-N})$
  - The decoder takes a window of N previously generated output tokens as additional inputs to make better predictions.
- Challenges
  - How can we design encoders and decoders as neural networks?
  - How to define loss functions to train models?
  - How to obtain data for training?