https://haoailab.com/cse291-s26/

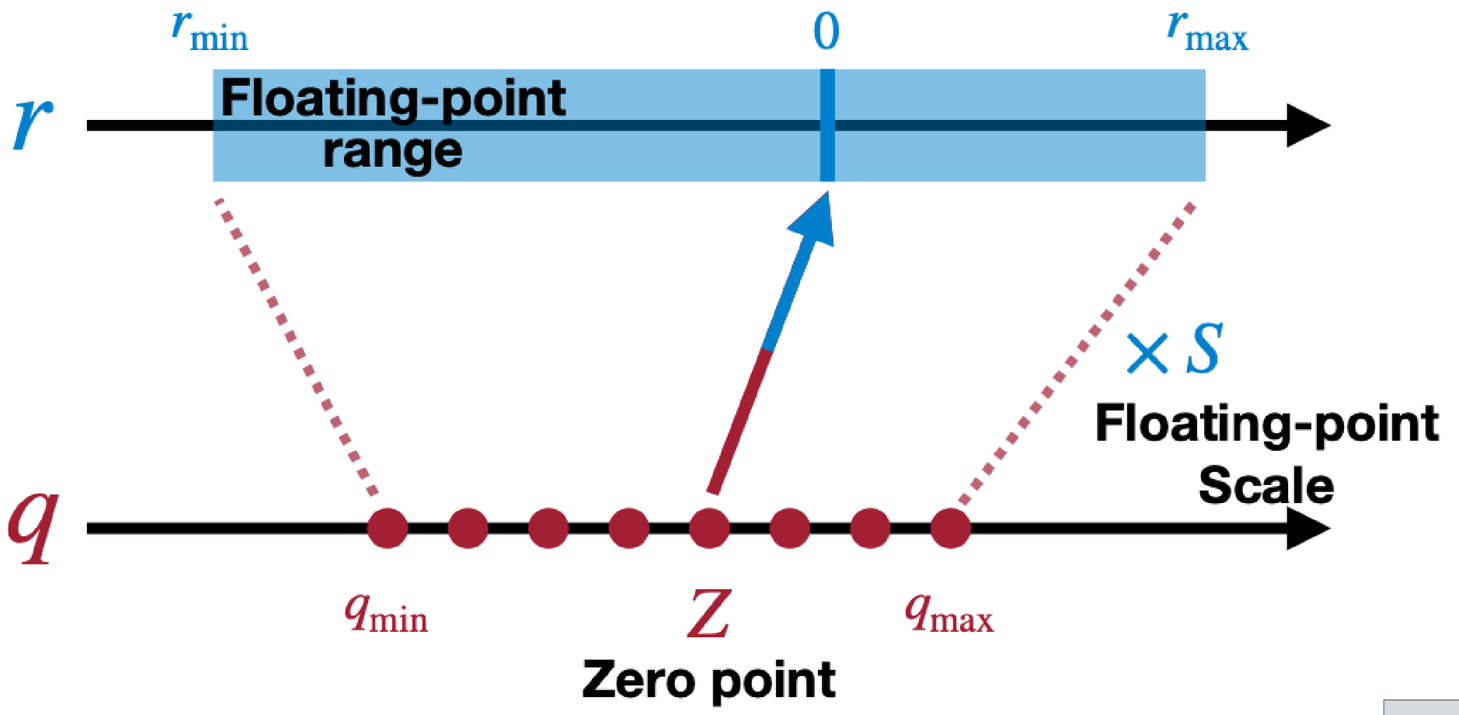
CSE/DSC 291: Deep Learning Systems
Spring 2026

LLM, diffusion, and case studies

Optimizations and Parallelization

Basics

$r = S(q - Z)$: Geometric Interpretation

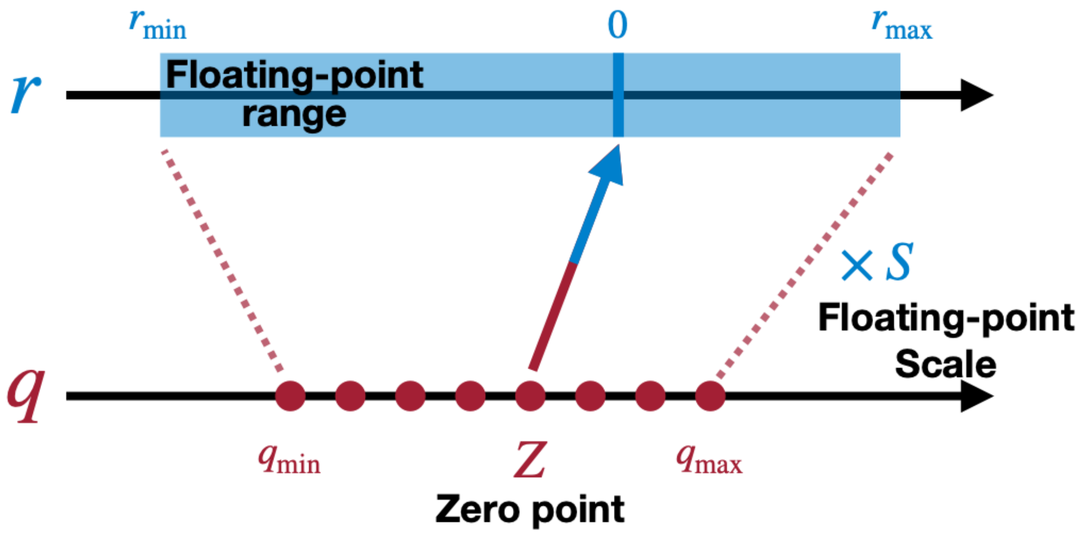


Bit Width	q_{min}	q_{max}
2	-2	1
3	-4	3
4	-8	7
N	-2^{N-1}	$2^{N-1}-1$

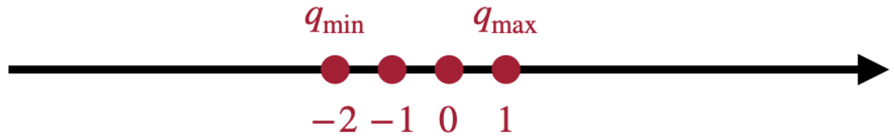
Q: How to determine S and Z?

$r = S(q - Z)$: Determine S and Z

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49



Binary	Decimal
01	1
00	0
11	-1
10	-2



$$Z = \text{round}(q_{min} - \frac{r_{min}}{S})$$

$$= \text{round}\left(-2 - \frac{-1.08}{1.07}\right) = -1$$

Apply Linear Quantization into Matmul

$$Y = WX$$

$$S_Y(q_Y - Z_Y) = S_W(q_W - Z_W)S_X(q_X - Z_X)$$

$$q_Y = \frac{S_W S_X}{S_Y}(q_W - Z_W)(q_X - Z_X) + Z_Y$$

$$q_Y = \frac{S_W S_X}{S_Y}(q_W q_X - Z_W q_X - Z_X q_W + Z_W Z_X) + Z_Y$$

Apply Linear Quantization into Matmul

$$q_Y = \frac{S_W S_X}{S_Y}(q_W q_X - Z_W q_X - Z_X q_W + Z_W Z_X) + Z_Y$$

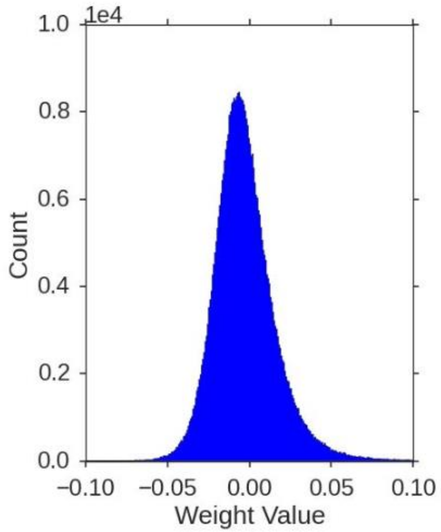
- precomputed;
- N-bit integer multiplication
- 32-bit integer addition/subtraction

- Empirically, $\frac{S_W S_X}{S_Y} \in (0, 1)$
- $\frac{S_W S_X}{S_Y} = 2^{-n} M_0, M_0 \in [0.5, 1)$ using fixed point multiplication and bit shift

Apply Linear Quantization into Matmul

$$q_Y = \frac{S_W S_X}{S_Y}(q_W q_X - Z_W q_X - Z_X q_W + Z_W Z_X) + Z_Y$$

- precomputed;
- N-bit integer multiplication
- 32-bit integer addition/subtraction



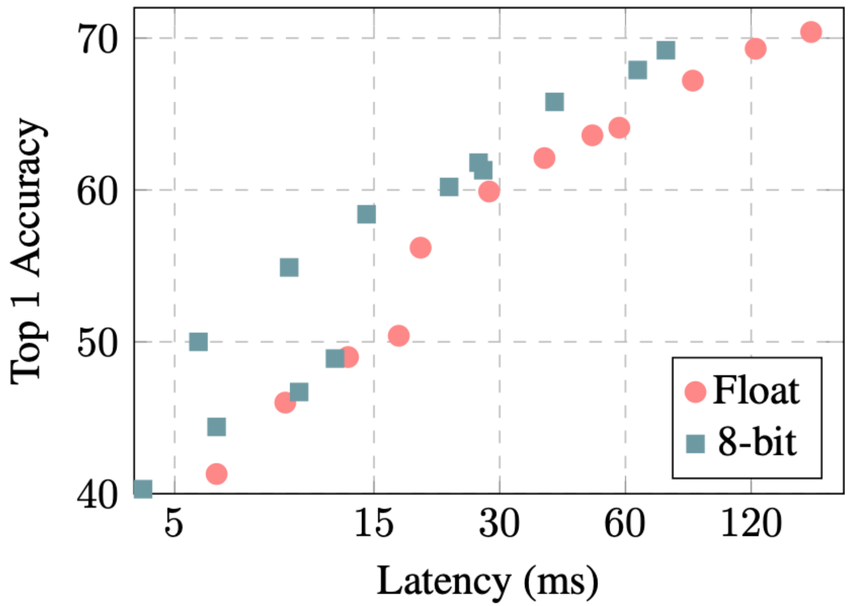
Empirically: $Z_W = 0$

$$q_Y = \frac{S_W S_X}{S_Y}(q_W q_X - Z_X q_W) + Z_Y$$

- Heavy lifting part
- integer multiplication

INT8 Linear Quantization Performance

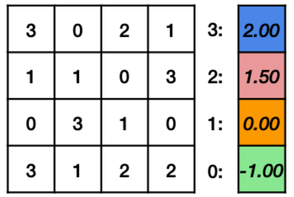
Neural Network	ResNet-50	Inception-V3
Floating-point Accuracy	76.4%	78.4%
8-bit Integer-quantized Acurracy	74.9%	75.4%



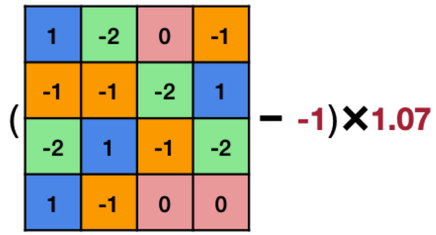
Latency-vs-accuracy tradeoff of float vs. integer-only MobileNets on ImageNet using Snapdragon 835 big cores.

Quantization Basics

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49



K-Means-based Quantization



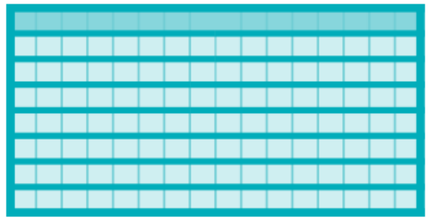
Linear Quantization

Storage	Floating point weights	integer weights; floating-point codebook	integer weights;
Compute	Floating point arithmetic	Floating point arithmetic	Integer arithmetic

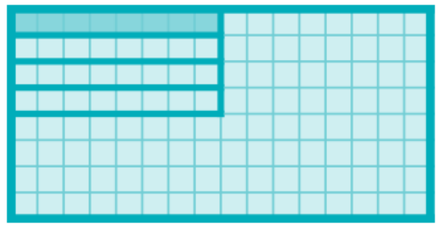
Quantization Granularity

- Per-tensor Quantization

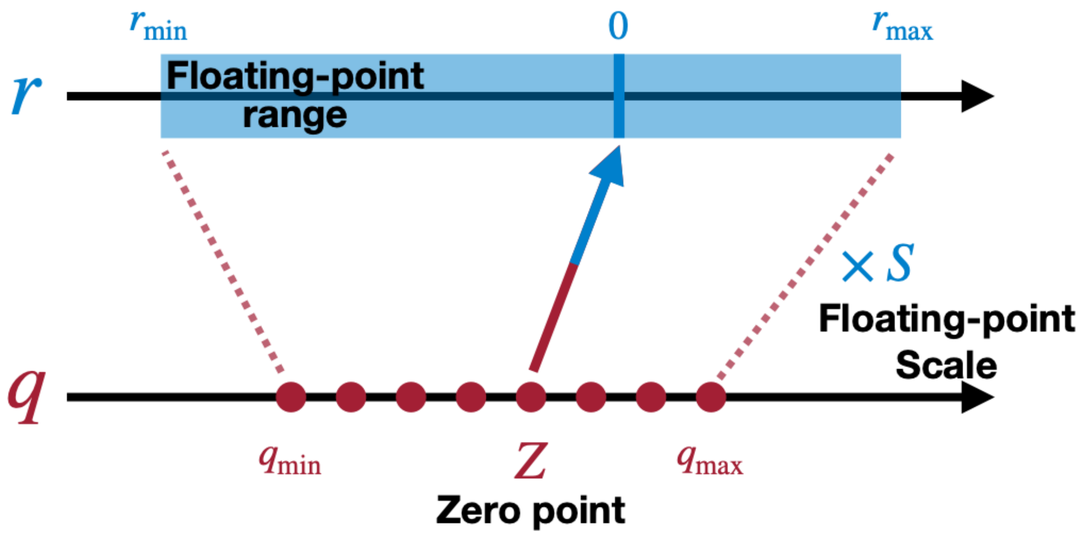
- Per-channel Quantization



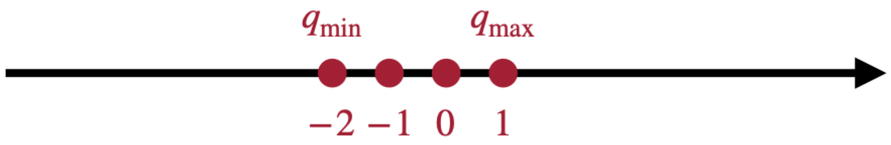
- Group Quantization



$r = S(q - Z)$: Determine S and Z



Binary	Decimal
01	1
00	0
11	-1
10	-2

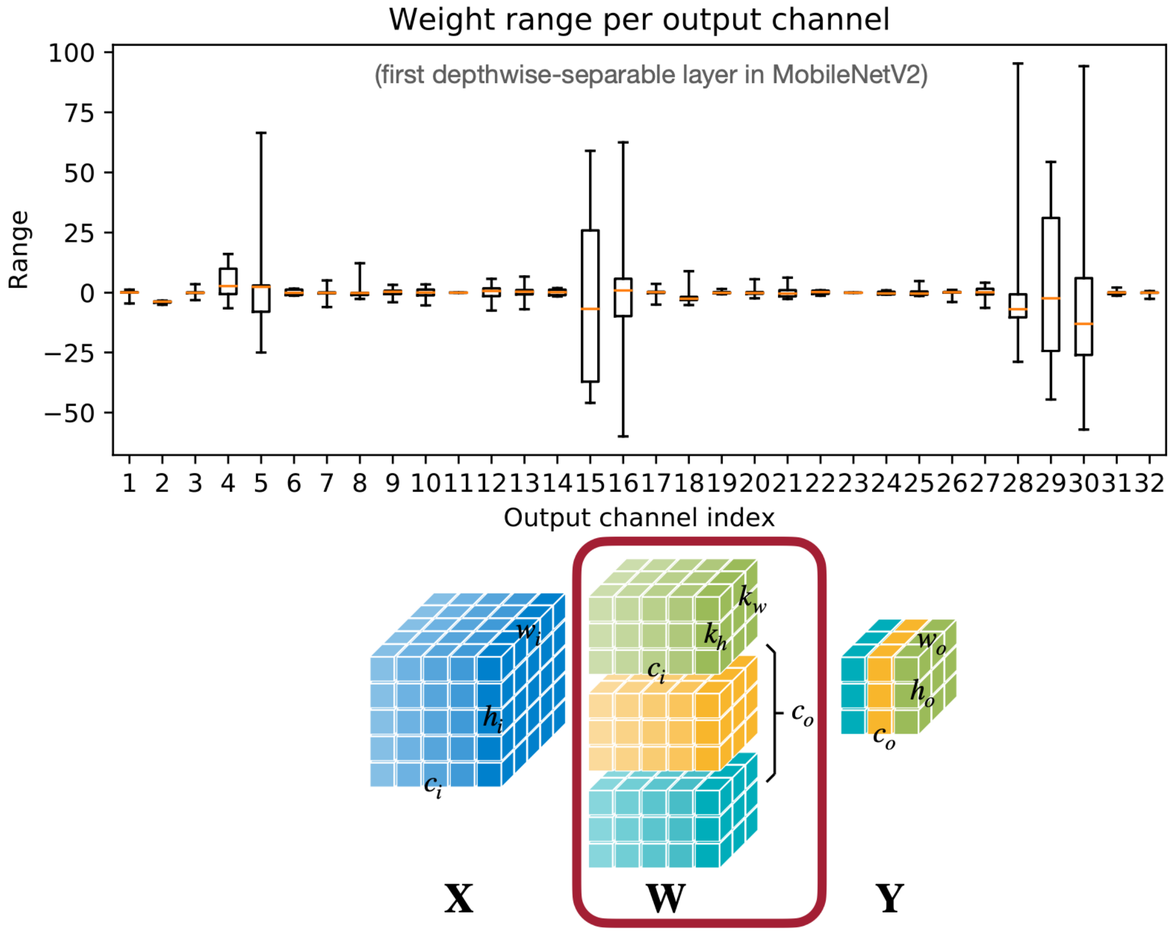


2.09	*-0.98*	*1.48*	*0.09*
0.05	*-0.14*	*-1.08*	*2.12*
-0.91	*1.92*	*0*	*-1.03*
1.87	*0*	*1.53*	*1.49*

Per-tensor quantization

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}} \qquad S = \frac{2.12 - (-1.08)}{1 - (-2)} = 1.07$$

Per-Tensor Quantization in Practice



- Per-tensor quantization
  - Using single scale S for whole weight tensor
- Common failure results from
  - Outlier weights
- Solution: per-channel quantization

Per-channel Quantization

- Example: 2-bit linear quantization

oc \ ic				
	2.09	*-0.98*	*1.48*	*0.09*
	0.05	*-0.14*	*-1.08*	*2.12*
	-0.91	*1.92*	*0*	*-1.03*
	1.87	*0*	*1.53*	*1.49*

Per-tensor quant

Per-channel quant

Per-channel Quantization

- Example: 2-bit linear quantization

ic

oc

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

Binary	Decimal
01	1
00	0
11	-1
10	-2

Per-tensor quant

$$|r|_{max} = 2.12$$

$$S = \frac{|r|_{max}}{q_{max}} = \frac{2.12}{2^{2-1} - 1} = 2.12$$

1	0	1	0
0	0	-1	1
0	1	0	0
1	0	1	1

Quantized

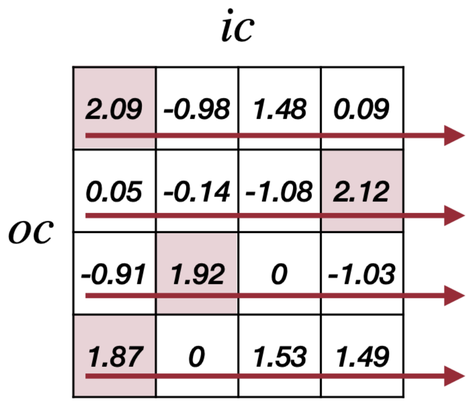
2.12	0	2.12	0
0	0	-2.12	2.12
0	2.12	0	0
2.12	0	2.12	2.12

Reconstructed

Per-channel quant

Per-channel Quantization

- Example: 2-bit linear quantization



Binary	Decimal
01	1
00	0
11	-1
10	-2

Per-tensor quant

$$|r|_{max} = 2.12$$

$$S = \frac{|r|_{max}}{q_{max}} = \frac{2.12}{2^{2-1}-1} = 2.12$$

1	0	1	0
0	0	-1	1
0	1	0	0
1	0	1	1

Quantized

2.12	0	2.12	0
0	0	-2.12	2.12
0	2.12	0	0
2.12	0	2.12	2.12

Reconstructed

$$\|W - S \odot q_W\| = 2.28$$

Per-channel quant

$$|r|_{max} = 2.09 \quad S_0 = 2.09$$
$$|r|_{max} = 2.12 \quad S_0 = 2.12$$
$$|r|_{max} = 1.92 \quad S_0 = 1.92$$
$$|r|_{max} = 1.87 \quad S_0 = 1.87$$

1	0	1	0
0	0	-1	1
0	1	0	-1
1	0	1	1

Quantized

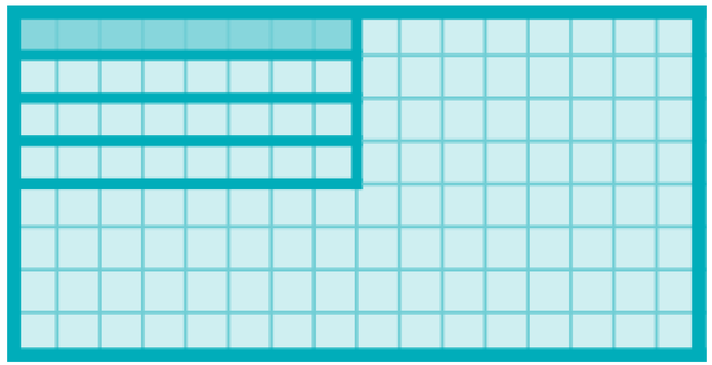
2.09	0	2.09	0
0	0	-2.12	2.12
0	1.92	0	-1.92
1.87	0	1.87	1.87

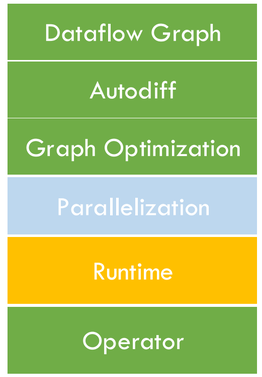
Reconstructed

$$\|W - S \odot q_W\| = 2.08$$

Group Quantization

- More fine-grained quantization granularity, e.g. per vector
- Pros: More accuracy, less quantization error
- Cons?

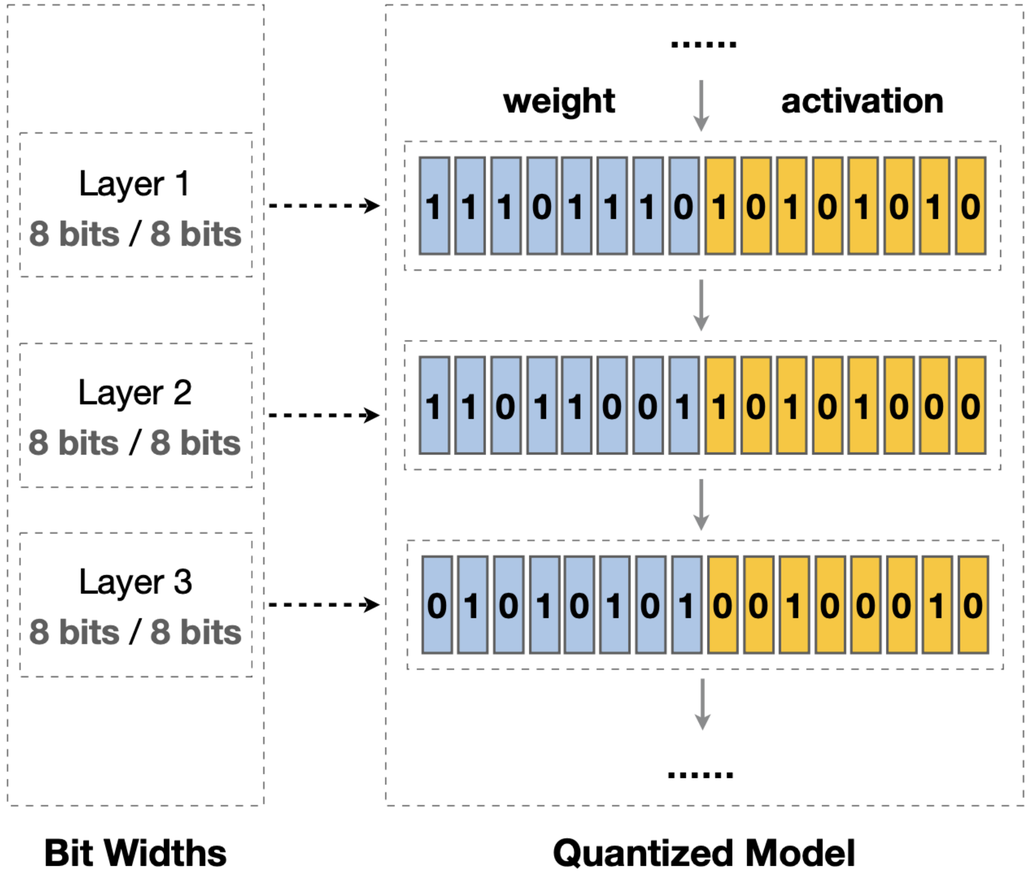




Today's Learning Goal

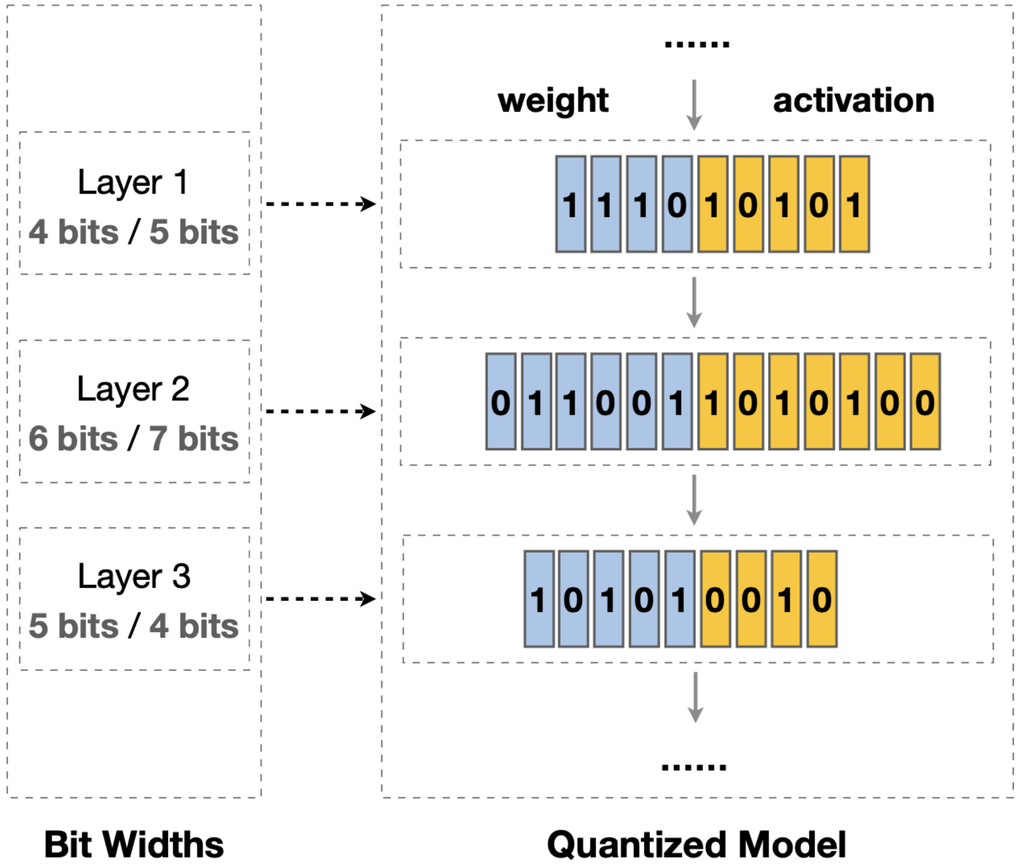
- Post-training Quantization
  - Quantization Granularity
  - Quantization on Activations
- **Mixed precision**
- Parallelization: Starter

Uniform Quantization

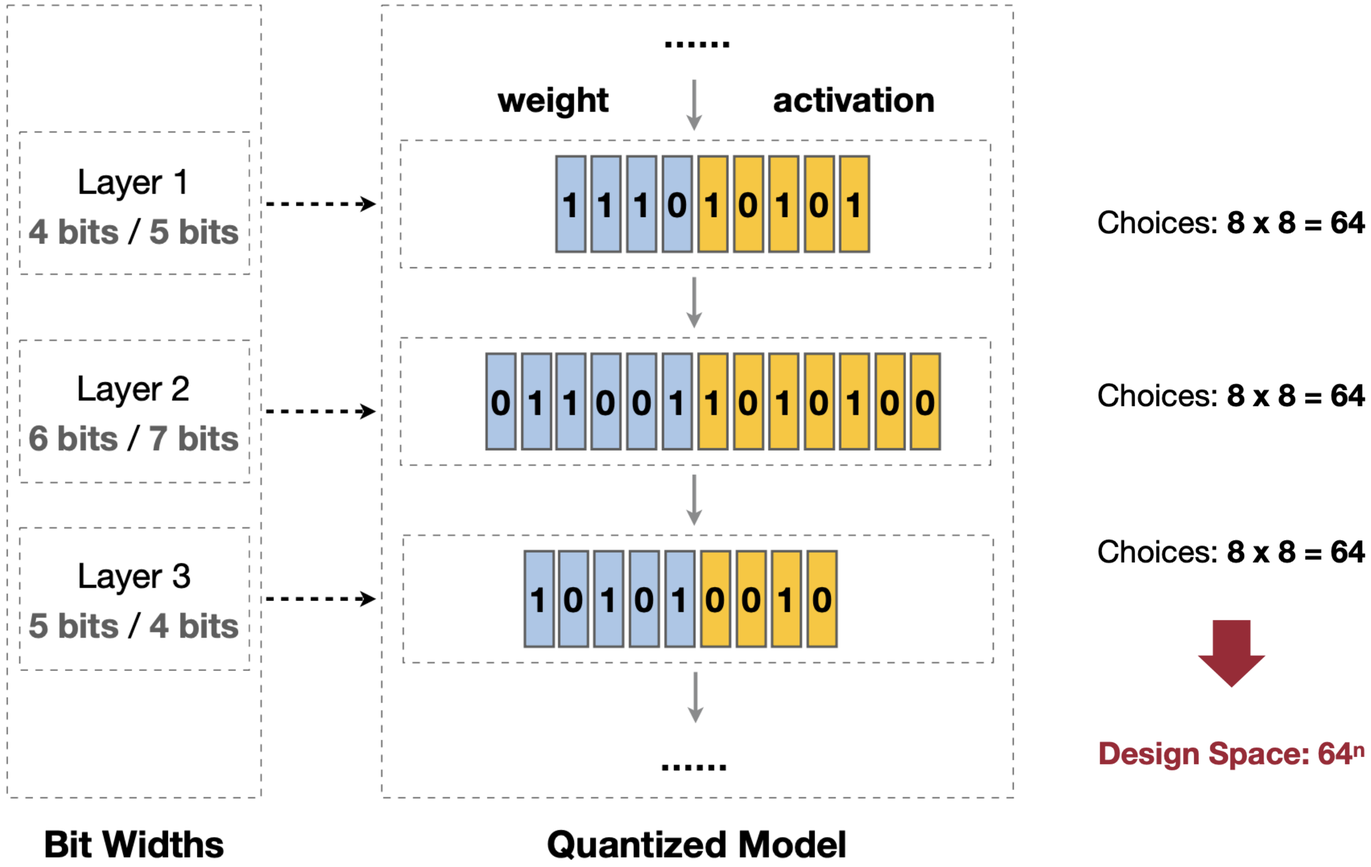


Mixed-Precision Quantization

- Intuition: why this works?



Mixed Precision: Design Space



In Practice: Mixed Precision Training

Published as a conference paper at ICLR 2018

MIXED PRECISION TRAINING

Sharan Narang*, **Gregory Diamos, Erich Elsen**†
Baidu Research
{sharan, gdiamos}@baidu.com

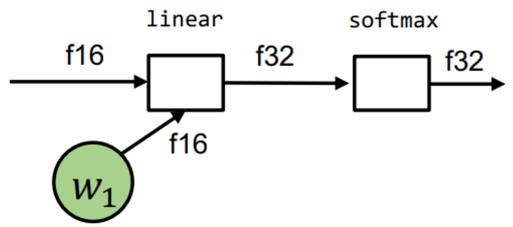
Paulius Micikevicius*, **Jonah Alben, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, Hao Wu**
NVIDIA
{pauliusm, alben, dagarcia, bginsburg, mhouston, okuchaiev, gavenkatesh, skyw}@nvidia.com

ABSTRACT

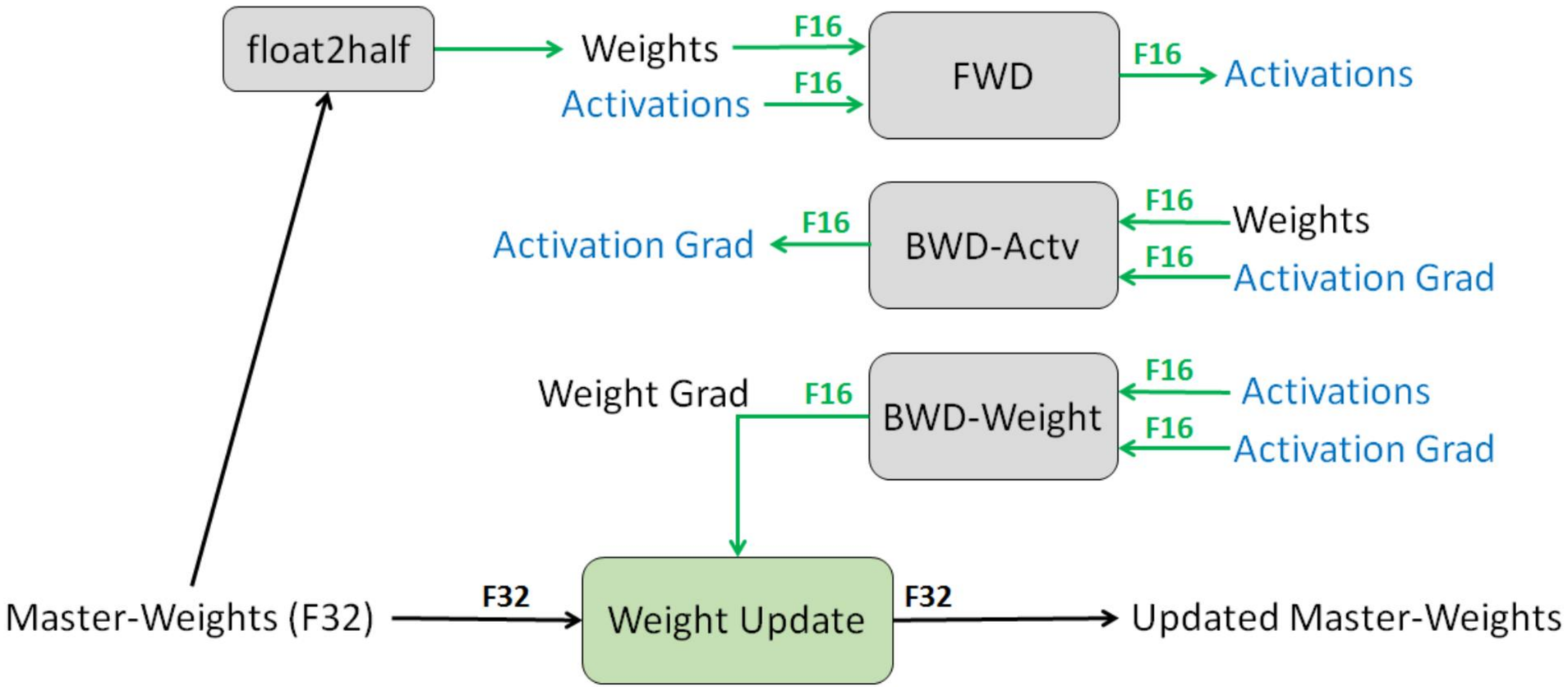
Increasing the size of a neural network typically improves accuracy but also increases the memory and compute requirements for training the model. We introduce methodology for training deep neural networks using half-precision floating point numbers, without losing model accuracy or having to modify hyper-parameters. This nearly halves memory requirements and, on recent GPUs, speeds up arithmetic. Weights, activations, and gradients are stored in IEEE half-precision format. Since this format has a narrower range than single-precision we propose three techniques for preventing the loss of critical information. Firstly, we recommend maintaining a single-precision copy of weights that accumulates the gradients after each optimizer step (this copy is rounded to half-precision for the forward- and back-propagation). Secondly, we propose loss-scaling to preserve gradient values with small magnitudes. Thirdly, we use half-precision arithmetic that accumulates into single-precision outputs, which are converted to half-precision before storing to memory. We demonstrate that the proposed methodology works across a wide variety of tasks and modern large scale (exceeding 100 million parameters) model architectures, trained on large datasets.

Mix-precision training

- Some layers are more sensitive to dynamic range/precision
  - Normalization: f / sum(f)
  - Softmax (same with normalization)
  - Param += \sum(grad_t) -> can loss precision during accum
- Idea: identify which ops are sensitive to precisions:
  - Use full precision (fp32) for them via upcasting
  - Use half precision to those robust ops

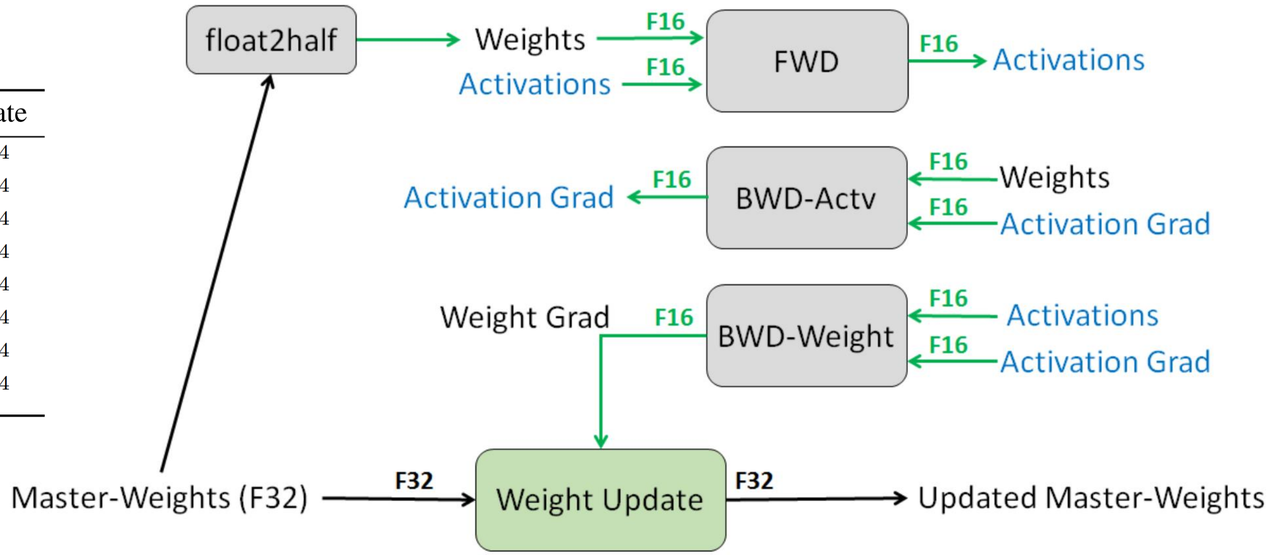


A standardized 16-32 mix-precision pipeline (Important!)



Analysis of the memory usage of Mix-precision training

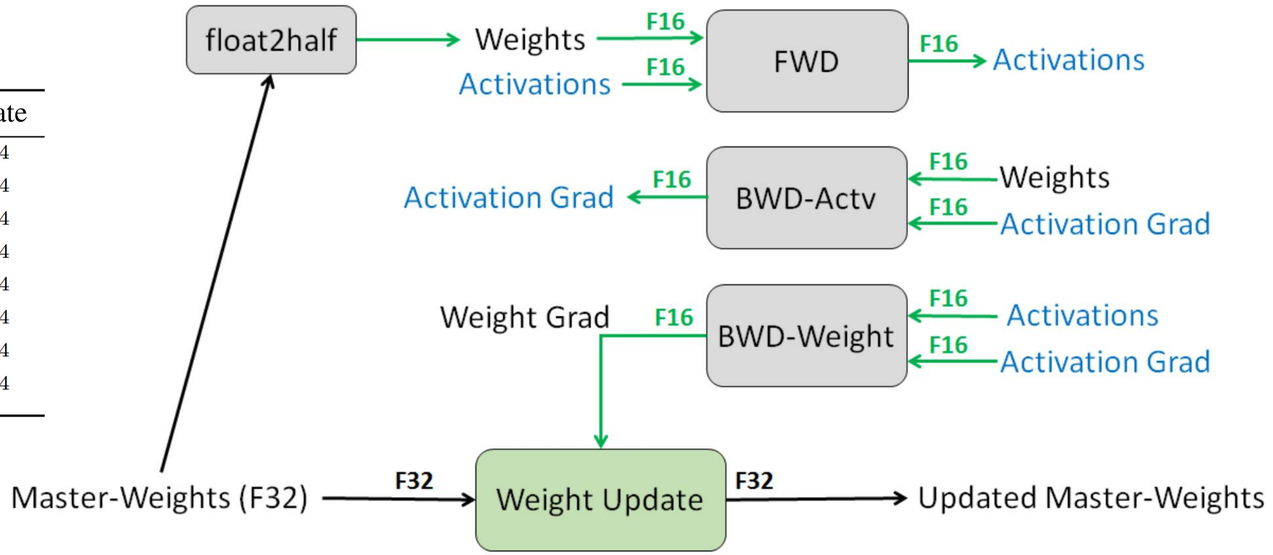
Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}



- Parameters: 175B * (fp16, 2 bytes) = 350G
- Assume we checkpoint at transformer layer boundary:
  - Activations: (N = 96) * 3.2M * 12288 * 2 = 7488 G
- How about optimizer states?

Analysis of the memory usage of Mix-precision training

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}



- How about optimizer states?
  - Master copy (fp32) = 4 * 175 = 700
  - Grad (fp16) = 2 * 175 = 350
  - Running copy (fp16) = 2 * 175 = 350
  - Adam mean and variance (fp32) = 2 * 4 * 175 = 1400G
- **Rule the thumb: (4 + 2 + 2 + 4 + 4) N = 16N memory for an LLM**

More on Tiny ML

- Running ML on edge devices is always strongly demanded
- Market characteristics: *very fragmented*
- Research directions: quantization, pruning, ML energy efficiency, federated ML etc.

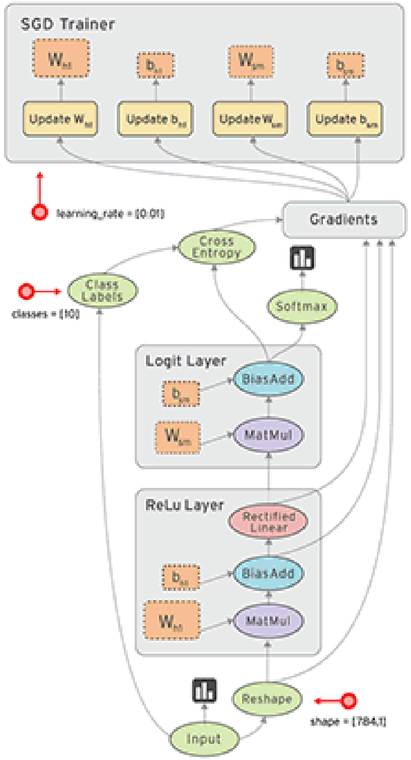
Zhijian Liu

zhijian [at] ucsd (dot) edu

I am a research scientist at NVIDIA and an incoming assistant professor at UCSD. I finished my PhD at MIT, advised by Song Han. My research focuses on efficient machine learning and systems.

I will be recruiting PhD students from HDSI/CSE in the Fall 2024 cycle and also looking for RAs/interns!

Put On Hold (we'll come back to Parallelization soon)



Dataflow Graph

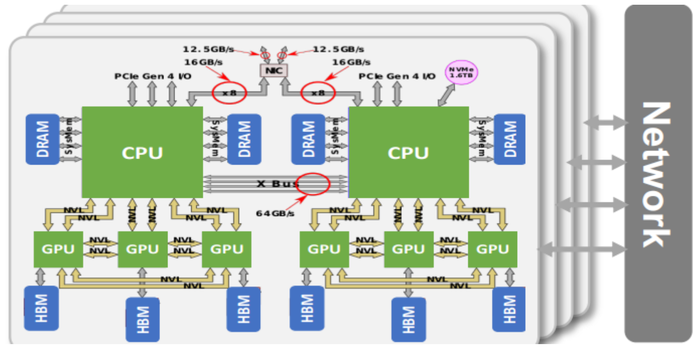
Autodiff

Graph Optimization

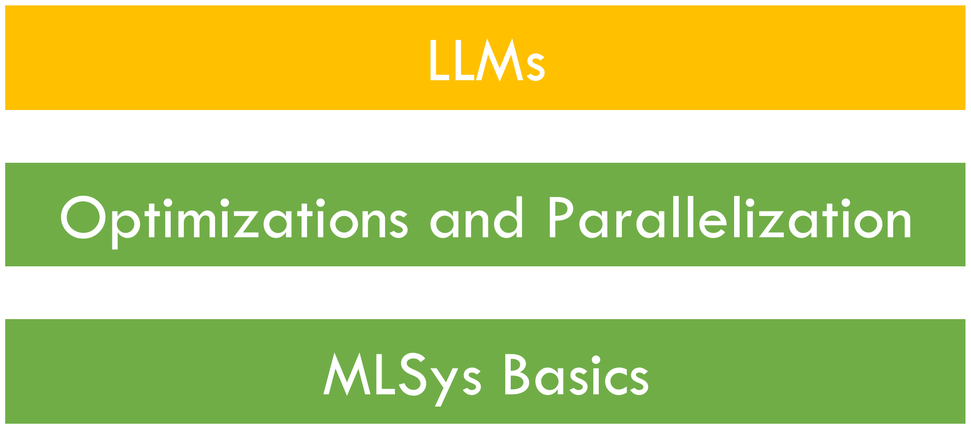
Parallelization

Runtime: schedule / memory

Operator optimization/compilation



Next: Connecting the Dots



Large Language Models

- Transformers, Attentions
- Scaling Law
- Serving and inference
- Training

Next Token Prediction

$$P(next\ word \mid prefix)$$

San Diego has very nice _	surfing	0.4
	weather	0.5
	snow	0.01
San Francisco is a city of _	innovation	0.6
	homeless	0.3

Next Token Prediction

Probability("San Diego has very nice weather")
= P("San Diego") P("has"|"San Diego")P("very"|"San Diego has")P("city"|…)…P("weather"|…)

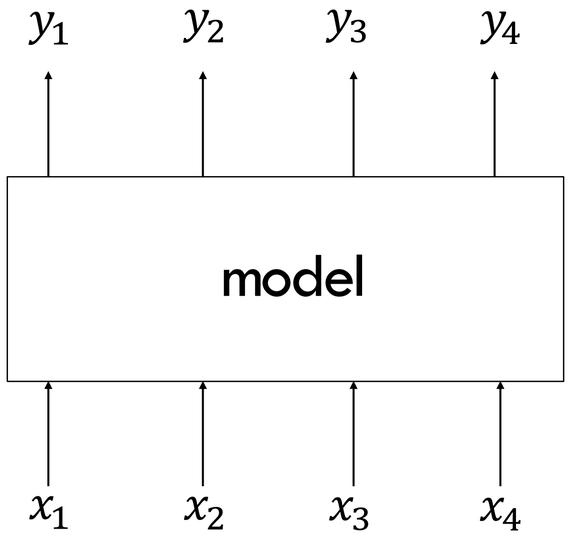
$$\text{Max}\, Prob(x_{1:T}) = \prod_{t=1}^{T} P(x_{t+1}|x_{1\ldots t})$$

MLE on observed data $x_{1:T}$,

This is next token prediction.
Predicting using seq2seq NNs.

Sequence Prediction

Take a set of input sequence, predict the output sequence



$y_t = f_\theta(x_{1:t})$

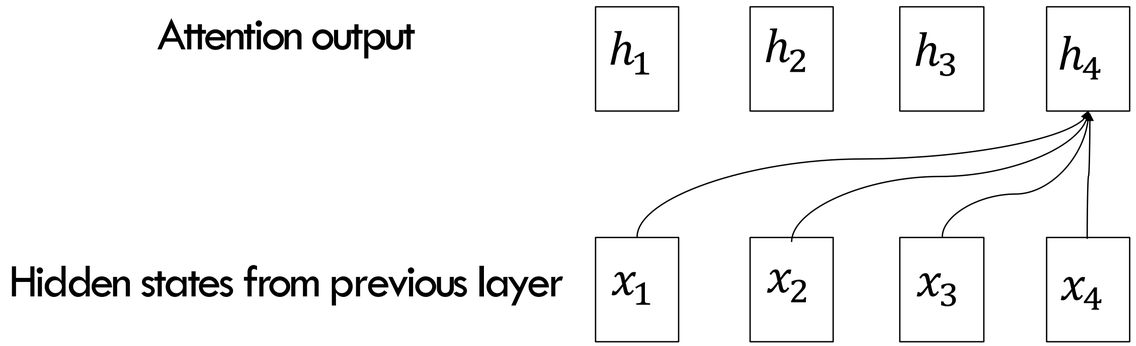
$$\ldots \quad \prod_{t=1}^{T} P(x_{t+1}|x_{1\ldots t})$$

Predict each output based on history

There are many ways to build up the predictive model

“Attention” Mechanism

Generally refers to the approach that weighted combine individual states



$$h_t = \sum_{i=1}^{t} s_i x_t$$

Intuitively s_i is “attention score” that computes how relevant the position i’s input is to this current hidden output

There are different methods to decide how attention score is being computed

Self-Attention Operation

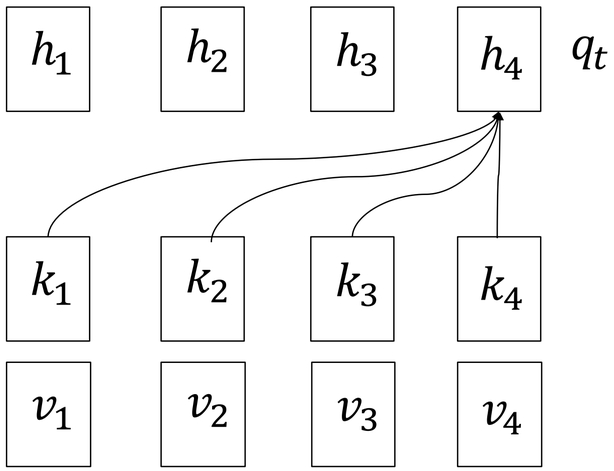
Self attention refers to a particular form of attention mechanism.
Given three inputs $Q, K, V \in \mathbb{R}^{T \times d}$ ("queries", "keys", "values")

Define the self-attention as:

$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{d^{1/2}}\right)V$$

A Closer Look at Self-Attention

Use q_t, k_t, v_t to refers to row t of the K matrix



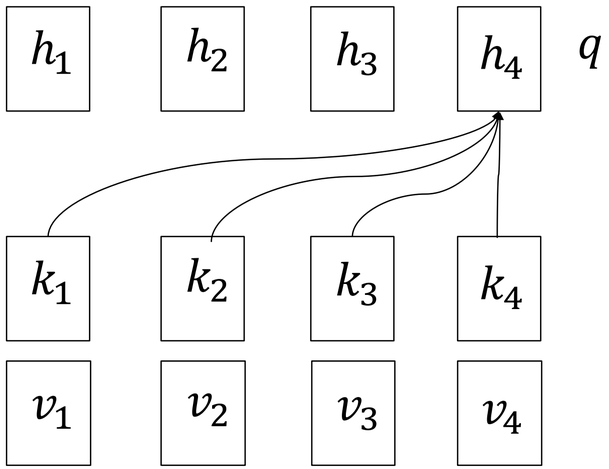
Ask the following question:

How to compute the output h_t, based on q_t, K, V one timestep t

To keep presentation simple, we will drop suffix t and just use q to refer to q_t in next few slide

A Closer Look at Self-Attention

Use q_t, k_t, v_t to refers to row t of the K matrix



Conceptually, we compute the output in the following two steps:

Pre-softmax "attention score"

$$s_i = \frac{1}{\sqrt{d}} q k_i^T$$

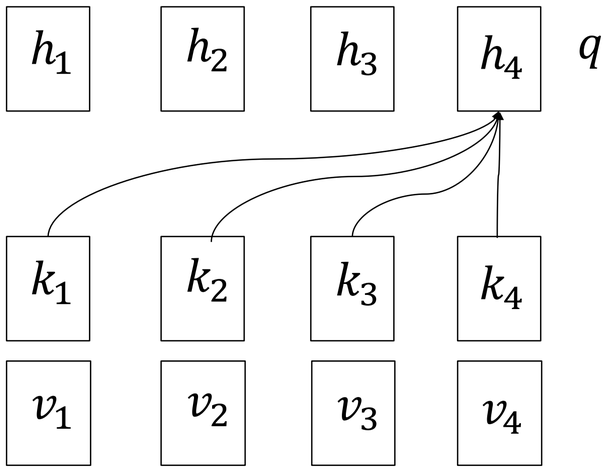
Weighed average via softmax

$$h = \sum_i \text{softmax}(s)_i v_i = \frac{\sum_i \exp(s_i) v_i}{\sum_j \exp(s_j)}$$

Intuition: s_i computes the relevance of k_i to the query q,
then we do weighted sum of values proportional to their relevance

Comparing the Matrix Form and the Decomposed Form

Use q_t, k_t, v_t to refers to row t of the K matrix



$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{d^{1/2}}\right)V$$

Pre-softmax "attention score"

$$S_{ti} = \frac{1}{\sqrt{d}} q_t k_i^T$$

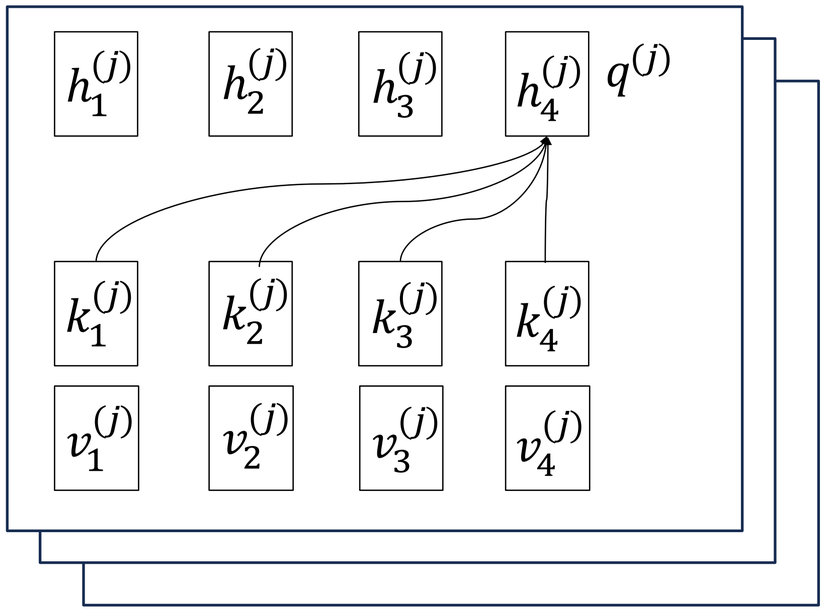
Weighed average via softmax

$$h_t = \sum_i \text{softmax}(S_{t,:})_i v_i = \text{softmax}(S_{t,:})V$$

Intuition: s_i computes the relevance of k_i to the query q,
then we do weighted sum of values proportional to their relevance

Multi-Head Attention

Have multiple "attention heads $Q^{(j)}, K^{(j)}, V^{(j)}$ denotes j-th attention head



Apply self-attention in each attention head

$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{d^{1/2}}\right)V$$

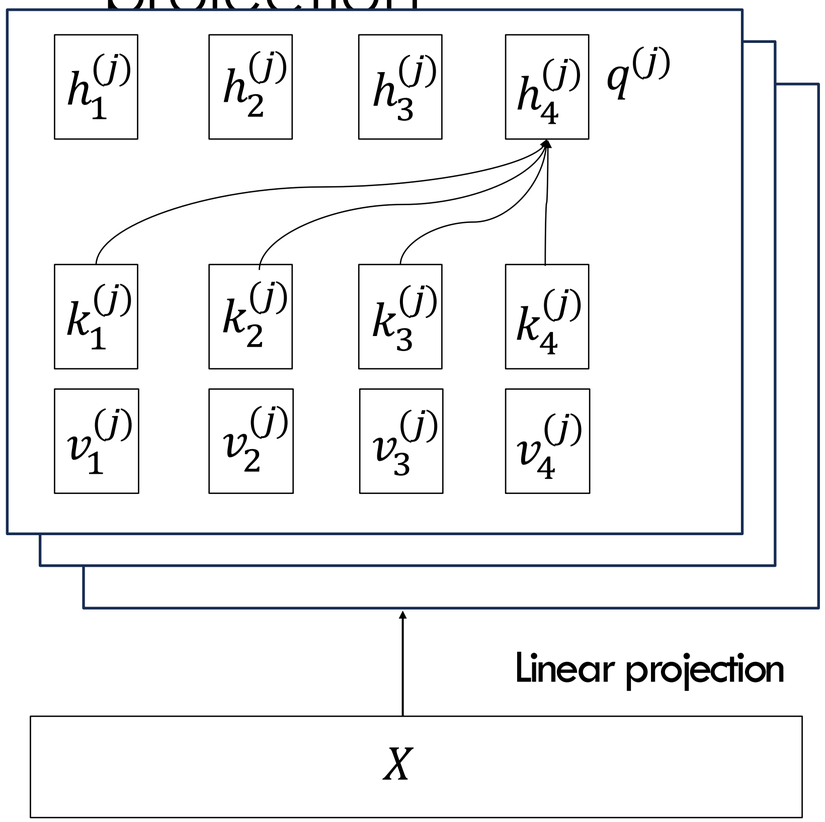
Concatenate all output heads together as output

Each head can correspond to different kind of information.

Sometimes we can share the heads: GQA(group query attention) all heads share K, V but have different Q

How to get Q K V?

Obtain Q, K, V from previous layer's hidden state X by linear projection



$$Q = XW_q$$
$$K = XW_K$$
$$V = XW_V$$

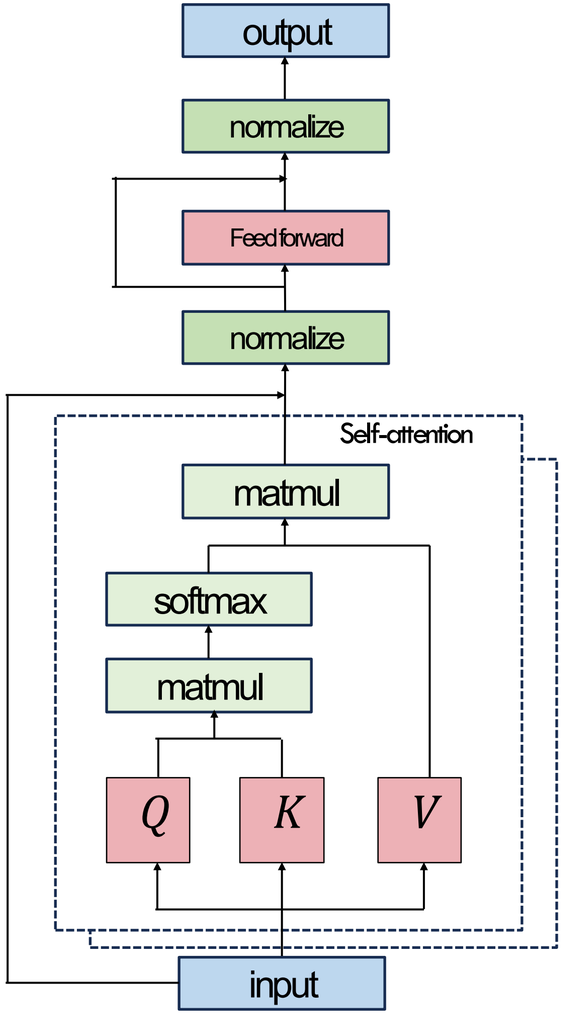
Can compute all heads and Q, K, V together then split/reshape out into individual Q, K, V with multiple heads

Transformer Block

A typical transformer block

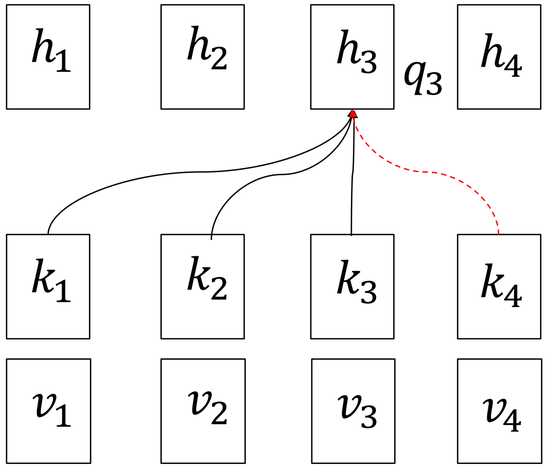
$$Z = \text{SelfAttention}(XW_K, XW_Q, XW_V)$$
$$Z = \text{LayerNorm}(X + Z)$$
$$H = \text{LayerNorm}(\text{ReLU}(ZW_1)W_2 + Z)$$

(multi-head) self-attention, followed by a linear layer and
ReLU and some additional residual connections and normalization



Masked Self-Attention

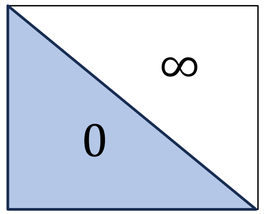
In the matrix form, we are computing weighted average over all inputs



In auto regressive models, usually it is good to maintain casual relation, and only attend to some of the inputs (e.g. skip the red dashed edge on the left). We can add "attention mask"

$$\text{MaskedSelfAttention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{d^{1/2}} - M\right)V$$

$$M_{ij} = \begin{cases} \infty, j > i \\ 0, j \leq i \end{cases}$$

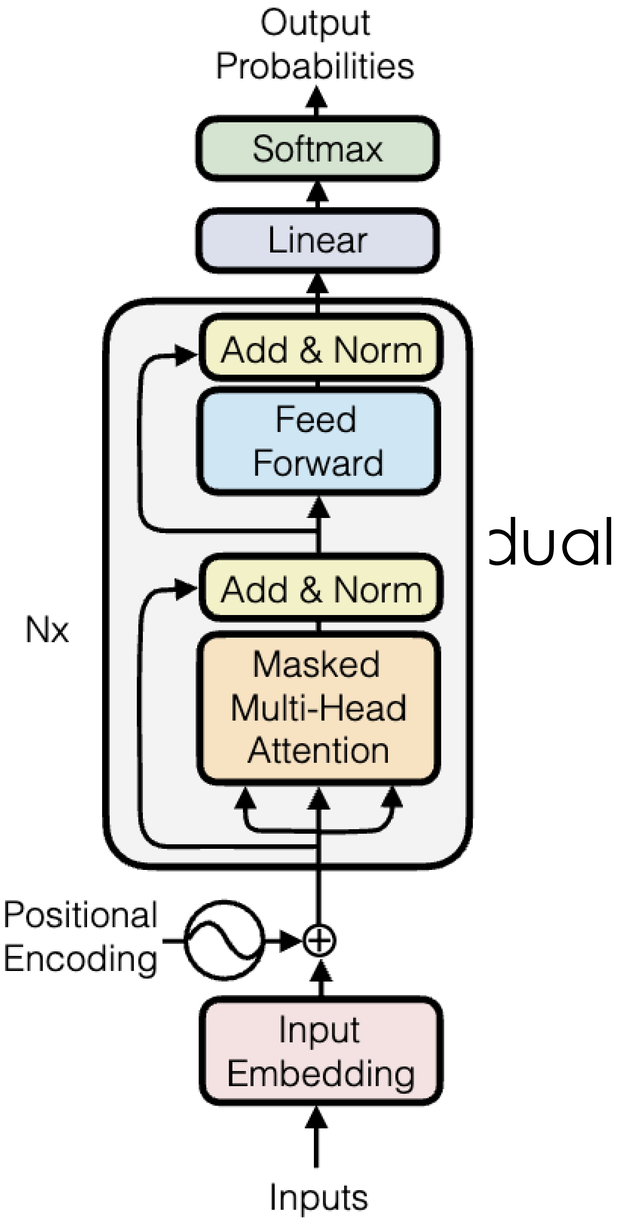


Only attend to previous inputs. Depending on input structure and model, attention mask can change.

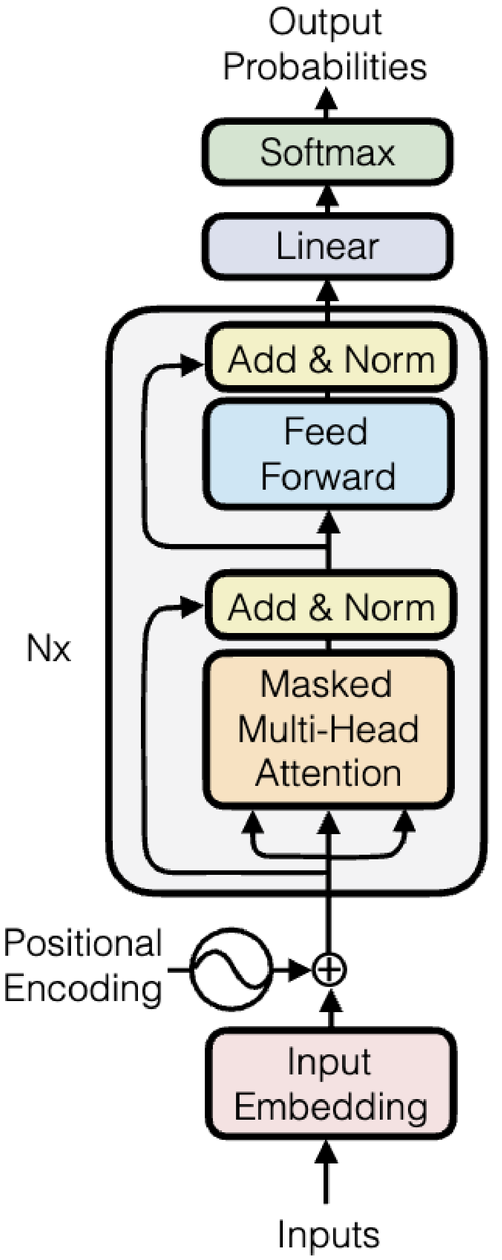
We can also simply skip the computation that are masked out if there is a special implementation to do so

Transformers

- Transformer decoders
  - Many of them
  - Really just: attentions + layernorm + MLPs - dual
- Word embeddings
- Position embeddings
  - Rotary embedding
- Loss function: cross entropy loss over a seque

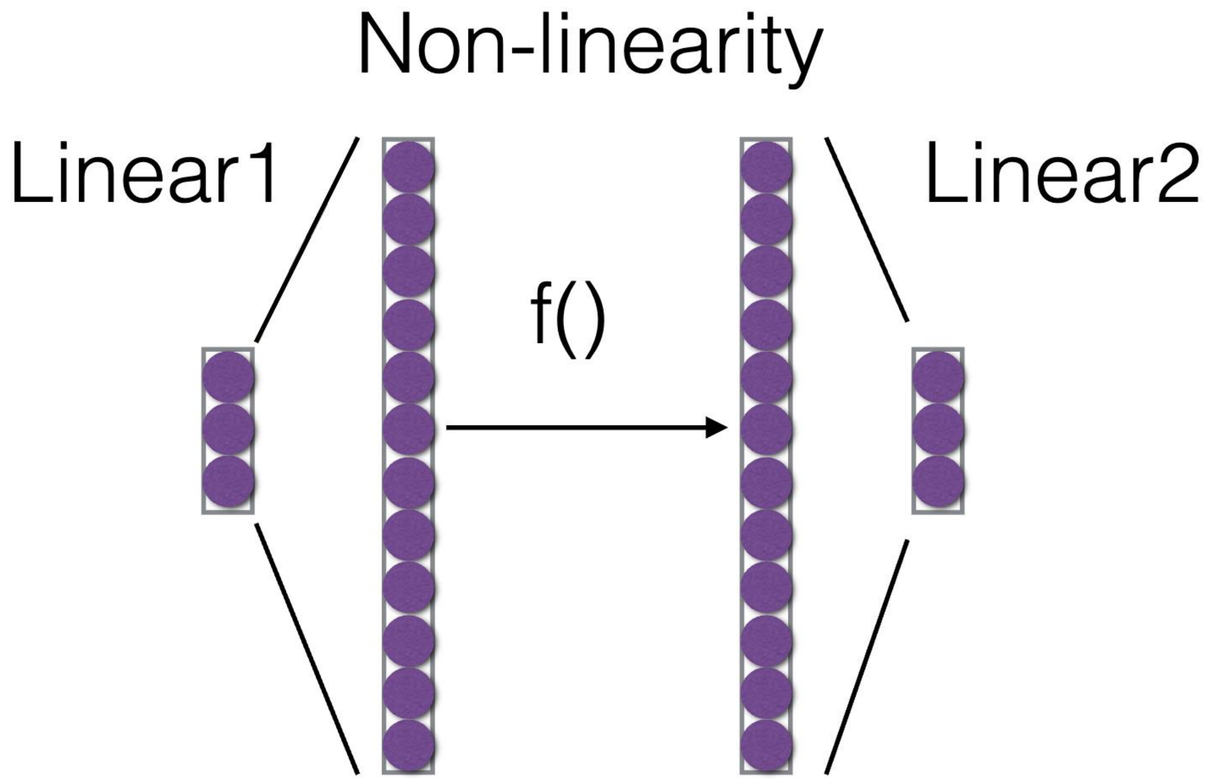


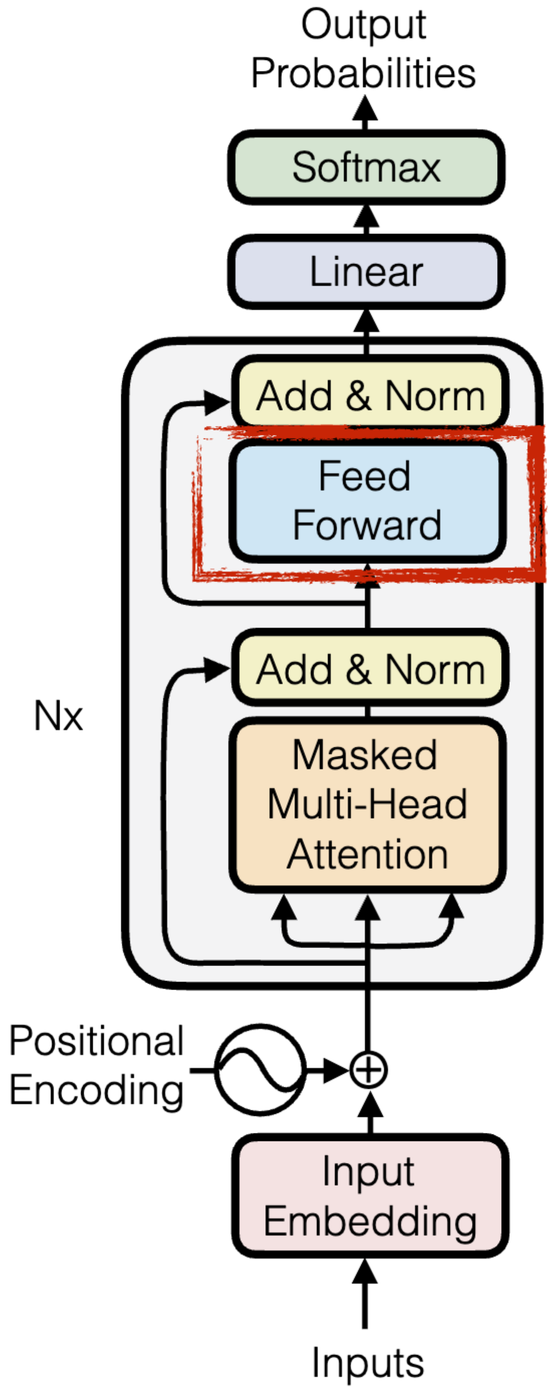
Transformers



Feedforward Layers

$$\mathrm{FFN}(x; W_1, \mathbf{b}_1, W_2, \mathbf{b}_2) = f(\mathbf{x}W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2$$



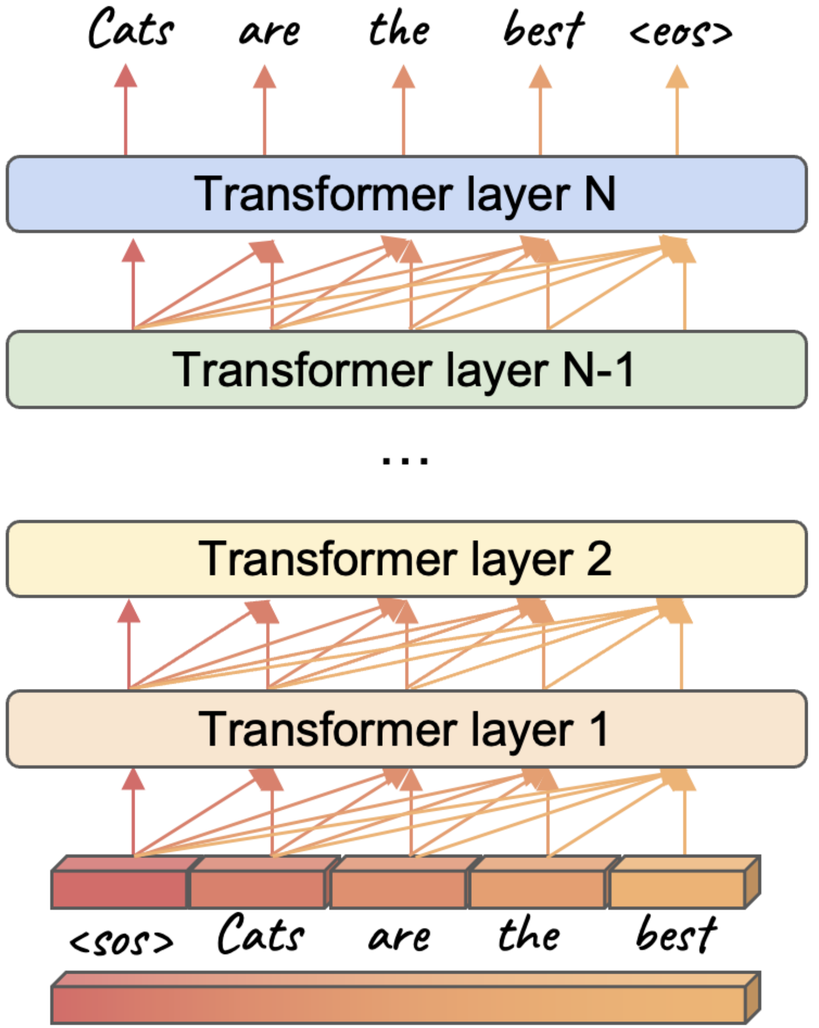


Computing Components in LLMs?

- Transformer decoders (many of them)
  - self-attentions (slow)
  - layernorm, residual (fast)
  - MLPs (slow)
  - Nonlinear (fast)
- Word embeddings (fast)
- Position embeddings (fast)
- Loss function: cross entropy loss over a sequence of words

Training LLMs

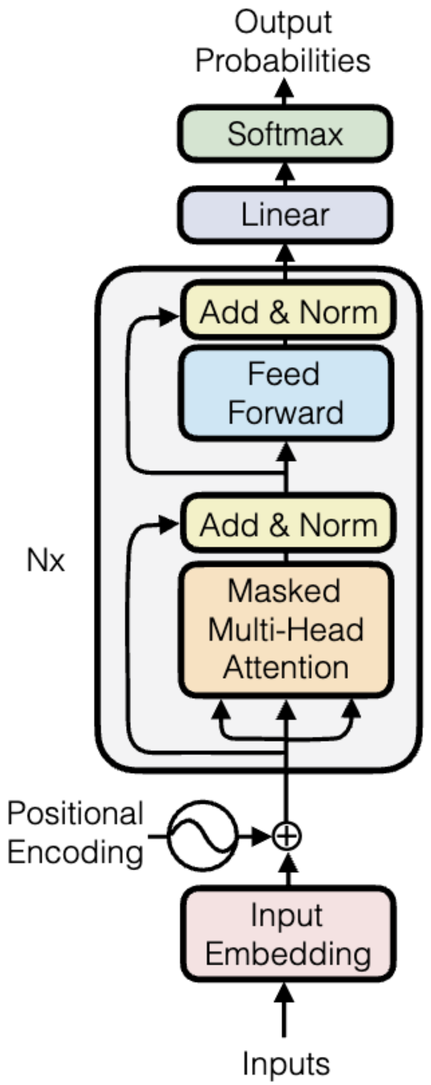
- Sequences are **known a priori**
- For each position, look at [1, 2, …, t-1] words to predict word t, and calculate the loss at t
- Parallelize the computation across all token positions, and then apply masking



Connecting the Dots: Compute/Comm characteristic of LLMs

Key characteristics: compute, memory, communication

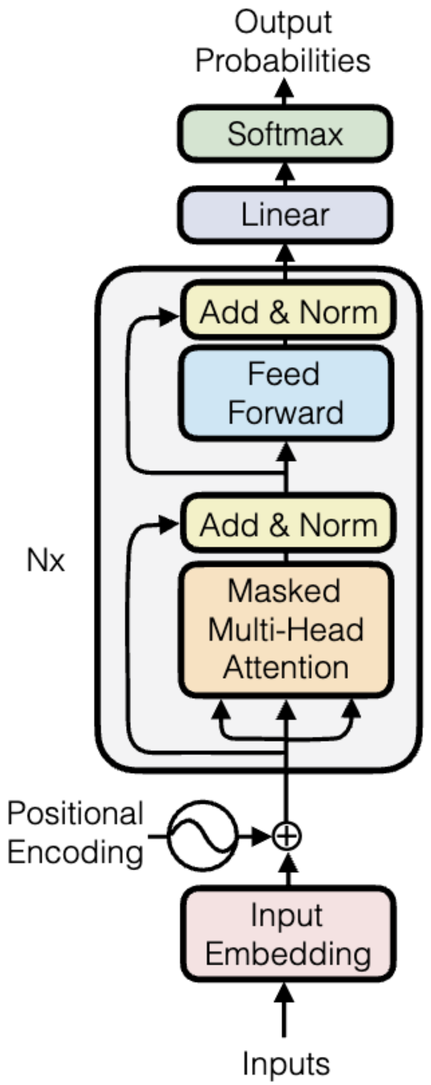
- calculate the number of parameters of an LLM?
  - memory, communication
- calculate the flops needed to train an LLM?
  - compute
- calculate the memory needed to train an LLM?
  - memory, communication

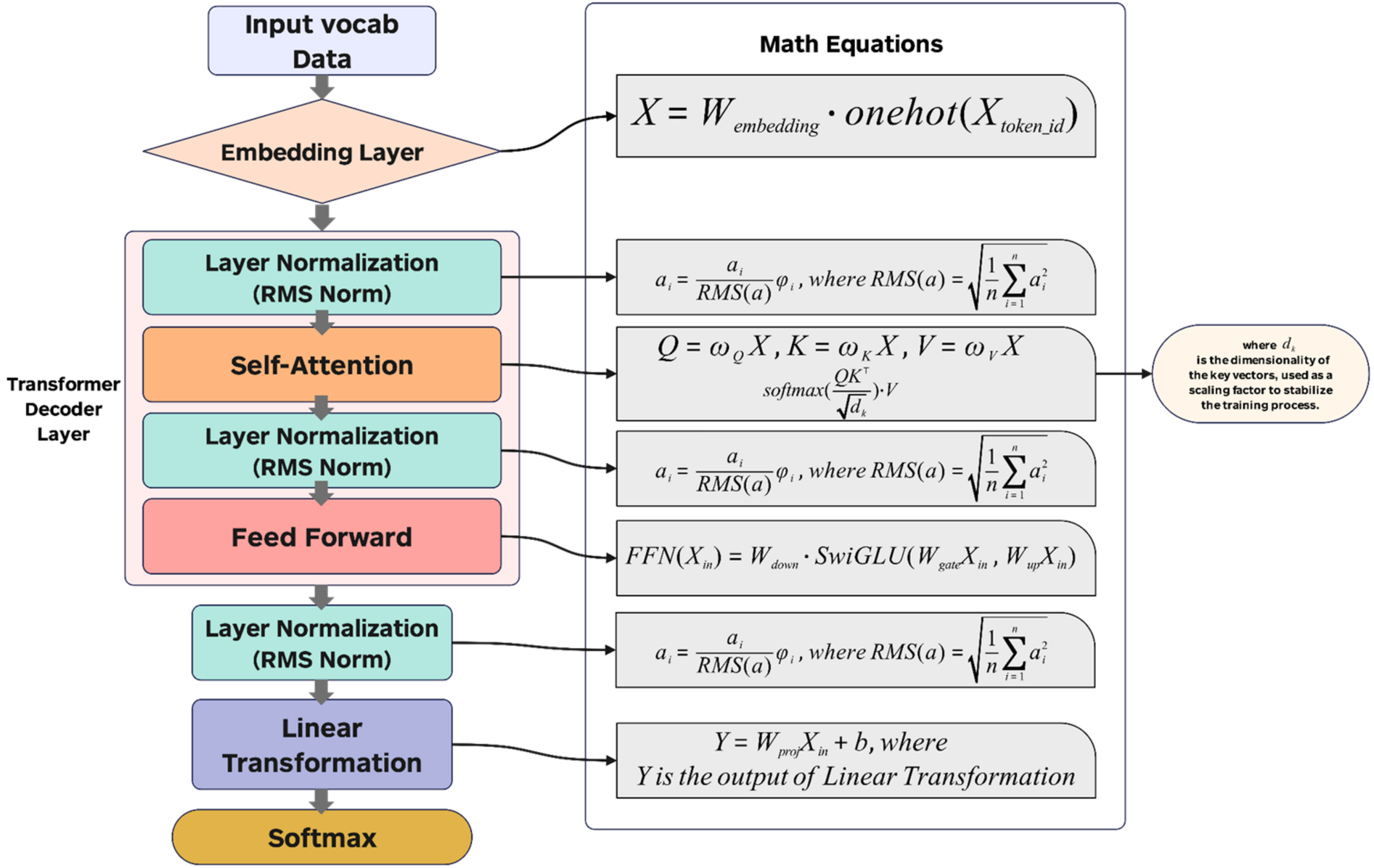


Connecting the Dots: Compute/Comm characteristic of LLMs

Key characteristics: compute, memory, communication

- calculate the number of parameters of an LLM?
- calculate the flops needed to train an LLM?
- calculate the memory needed to train an LLM?




Input vocab Data
Embedding Layer
Transformer Decoder Layer
Layer Normalization (RMS Norm)
Self-Attention
Layer Normalization (RMS Norm)
Feed Forward
Layer Normalization (RMS Norm)
Linear Transformation
Softmax
Math Equations
$X = W_{embedding} \cdot onehot(X_{token\_id})$
$a_i = \frac{a_i}{RMS(a)} \varphi_i$, where $RMS(a) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}$
$Q = \omega_Q X, K = \omega_K X, V = \omega_V X$
$softmax(\frac{QK^T}{\sqrt{d_k}}) \cdot V$
where d_k is the dimensionality of the key vectors, used as a scaling factor to stabilize the training process.
$a_i = \frac{a_i}{RMS(a)} \varphi_i$, where $RMS(a) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}$
$FFN(X_{in}) = W_{down} \cdot SwiGLU(W_{gate} X_{in}, W_{up} X_{in})$
$a_i = \frac{a_i}{RMS(a)} \varphi_i$, where $RMS(a) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}$
$Y = W_{proj} X_{in} + b$, where
Y is the output of Linear Transformation

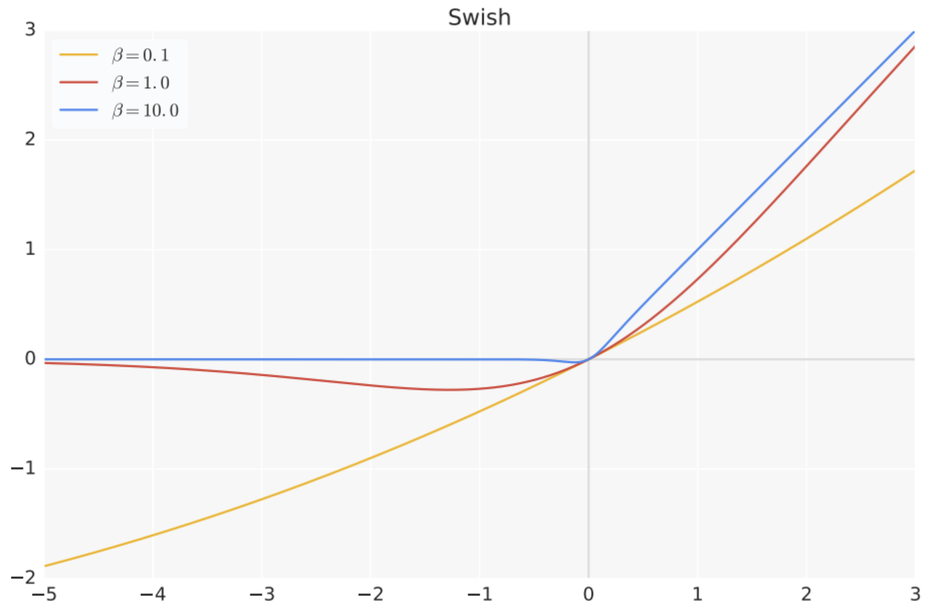
Feed Forward SwiGLU

The general formula for SwiGLU is:

$$\mathrm{SwiGLU}(x) = \mathrm{Swish}(xW_1 + b_1) \odot (xW_2 + b_2)$$

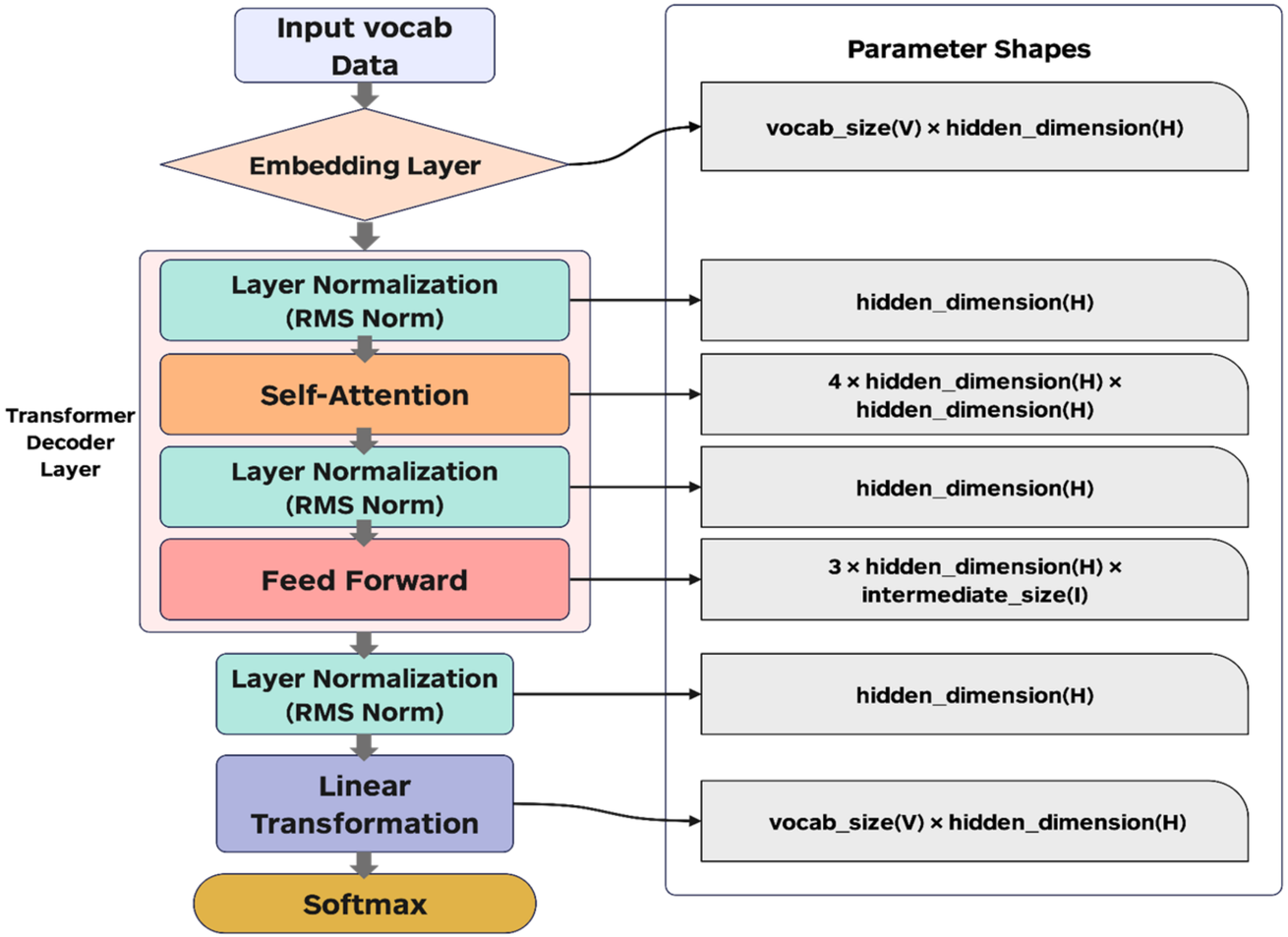
Swish is the activation function applied to one branch, defined as:

$$\mathrm{Swish}(z) = z \cdot \sigma(z)$$

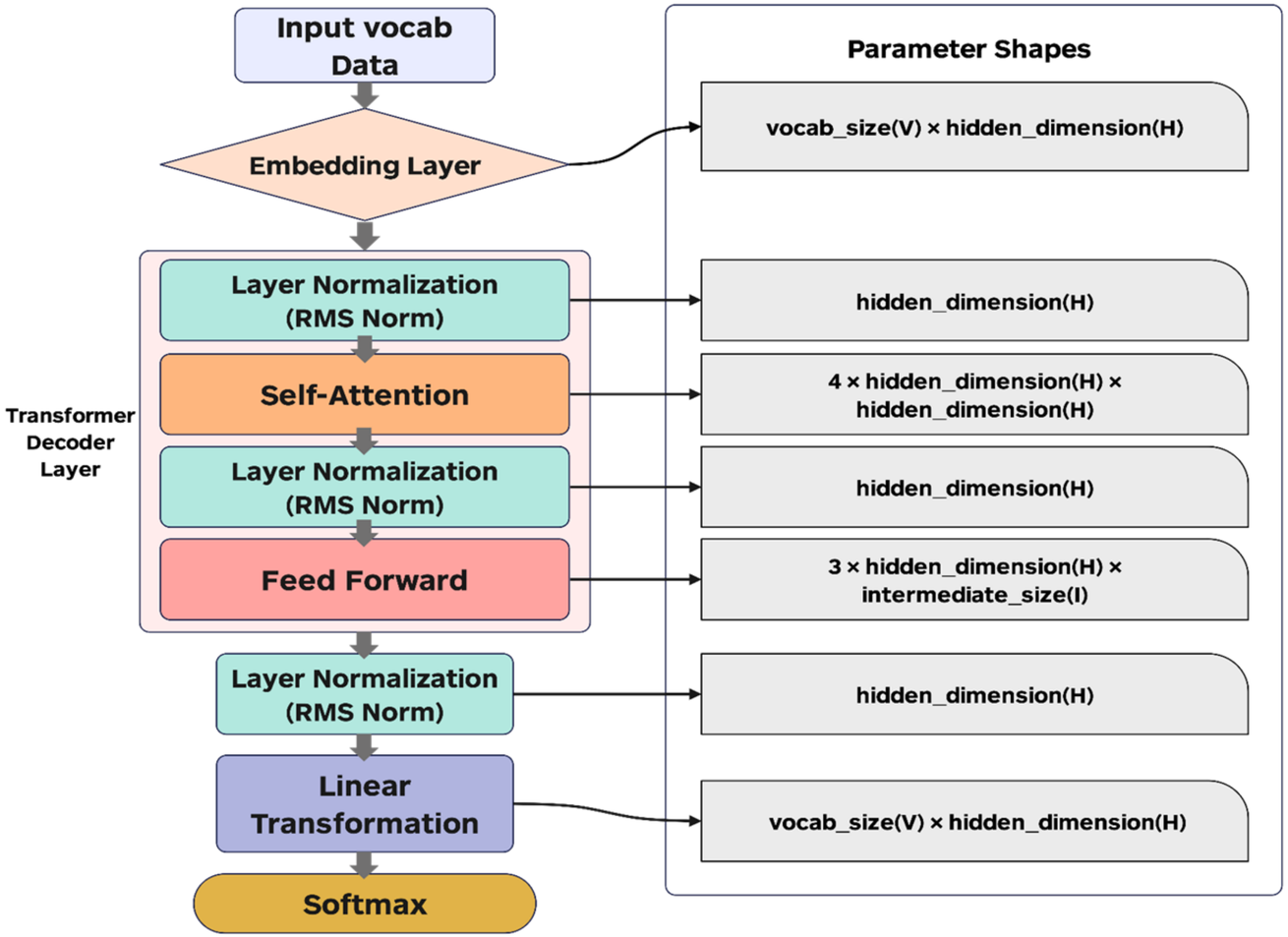


- SwiGLU helps the model capture more complex patterns by selectively gating information
- Swish is smoother than traditional activations ReLU

Summary


Input vocab Data
Embedding Layer
Transformer Decoder Layer
Layer Normalization (RMS Norm)
Self-Attention
Layer Normalization (RMS Norm)
Feed Forward
Layer Normalization (RMS Norm)
Linear Transformation
Softmax
Parameter Shapes
vocab_size(V) × hidden_dimension(H)
hidden_dimension(H)
4 × hidden_dimension(H) × hidden_dimension(H)
hidden_dimension(H)
3 × hidden_dimension(H) × intermediate_size(I)
hidden_dimension(H)
vocab_size(V) × hidden_dimension(H)

Scaling Up: Where is the Potential Bottleneck?

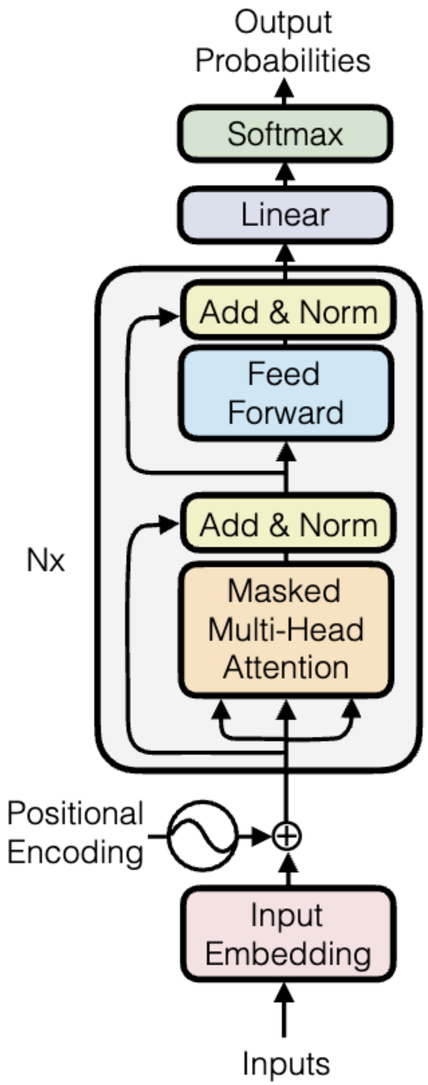


In PA3, you will implement this function😉

Connecting the Dots: Compute/Comm characteristic of LLMs

Key characteristics: compute, memory, communication

- calculate the number of parameters of an LLM?
- calculate the flops needed to train an LLM?
- calculate the memory needed to train an LLM?

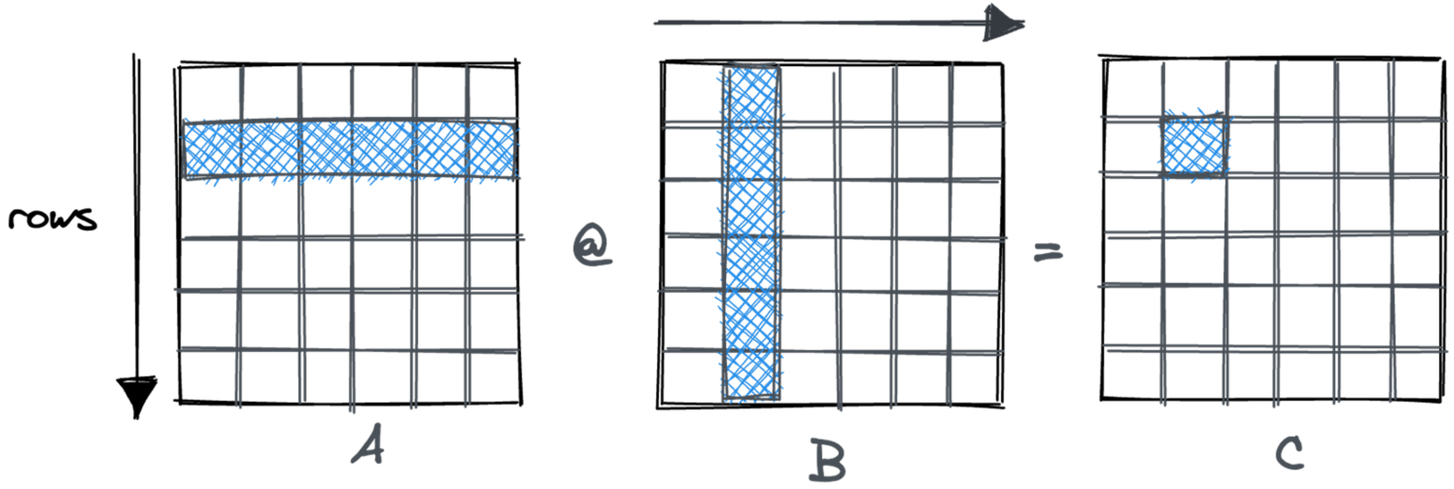


Estimate the Compute: FLOPs

The FLOPs for multiplying two matrices of dimensions m×n and n×h can be calculated as follows:

$$\text{FLOPs} = m \times h \times (2n - 1)$$

So the total number of FLOPs is roughly $\text{FLOPs} \approx 2m \times n \times h$



LLama 2 7B Flops Forward Calculation (Training)

Hyperparameters:

Batch size: b

Sequence length: s

The number of attention heads: n

Hidden state size of one head: d

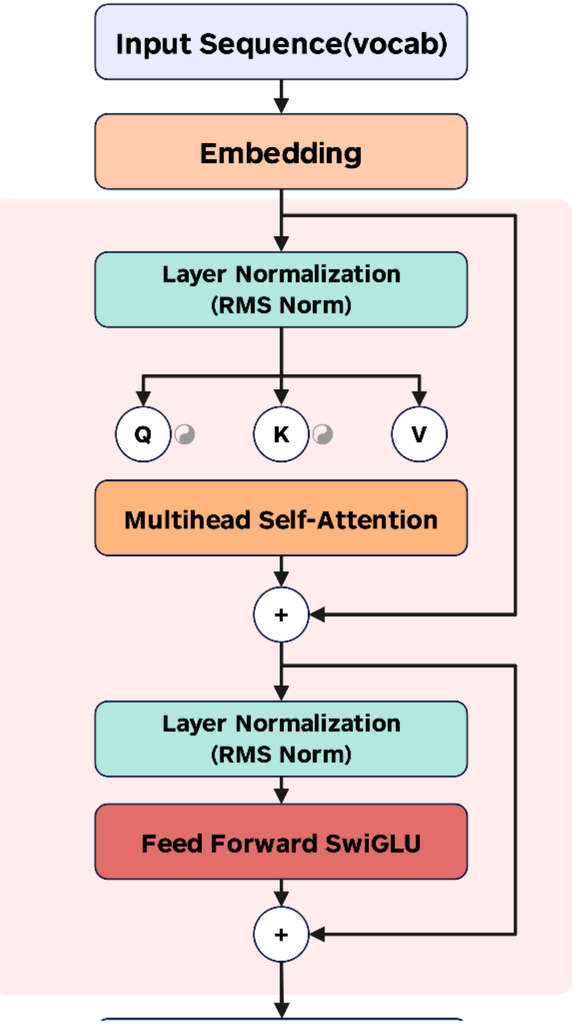
Hidden state size: h (h = n * d)

SwiGLU proj dim: i

Vocab size: v

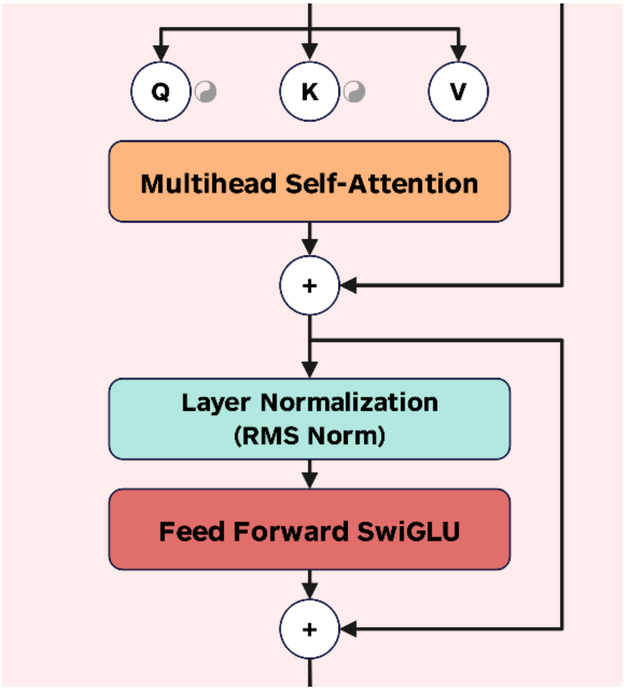
Input:	Output Shape:	FLOPs
X	(b, s, h)	0
Self Attention:		
XW_Q, XW_K, XW_V	(b, s, h)	$3 * 2bsh^2$
RoPE	(b, n, s, d)	$3bsnd$
$P = Softmax(QK^T/\sqrt{d})$	(b, n, s, s)	$2bs^2nd + 3bs^2n$
PV	(b, n, s, d)	$2bs^2nd$
AW_O	(b, s, h)	$2bsh^2$
Residual Connection:	(b, s, h)	bsh

Batch size: b
Sequence length: s
# of attention heads: n
Hidden state dim of one head: d
Hidden state dim: h



Batch size: b
Sequence length: s
Hidden state dim: h
SwiGLU proj dim: i

Output from Self Attn:	Output Shape:	FLOPs
X	(b, s, h)	
Feed-Forward		0
SwiGLU:	(b, s, i)	2 * 2bshi
XW_{gate}, XW_{up}	(b, s, i)	4bsi
Swish Activation	(b, s, i)	bsi
Element-wise *	(b, s, h)	2bshi
XW_{down}		4bsh + 2bs



RMS Norm:

$$\text{SwiGLU}(x) = \text{Swish}(xW_1 + b_1) \odot (xW_2 + b_2)$$

1. **Calculate Root Mean Square:**
   - $\text{RMS}(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}$
2. **Normalize:**
   - $\text{RMSNorm}(x) = \frac{x}{\text{RMS}(x)+\epsilon} \cdot \gamma$

LLama 2 7B Flops Forward (Training)

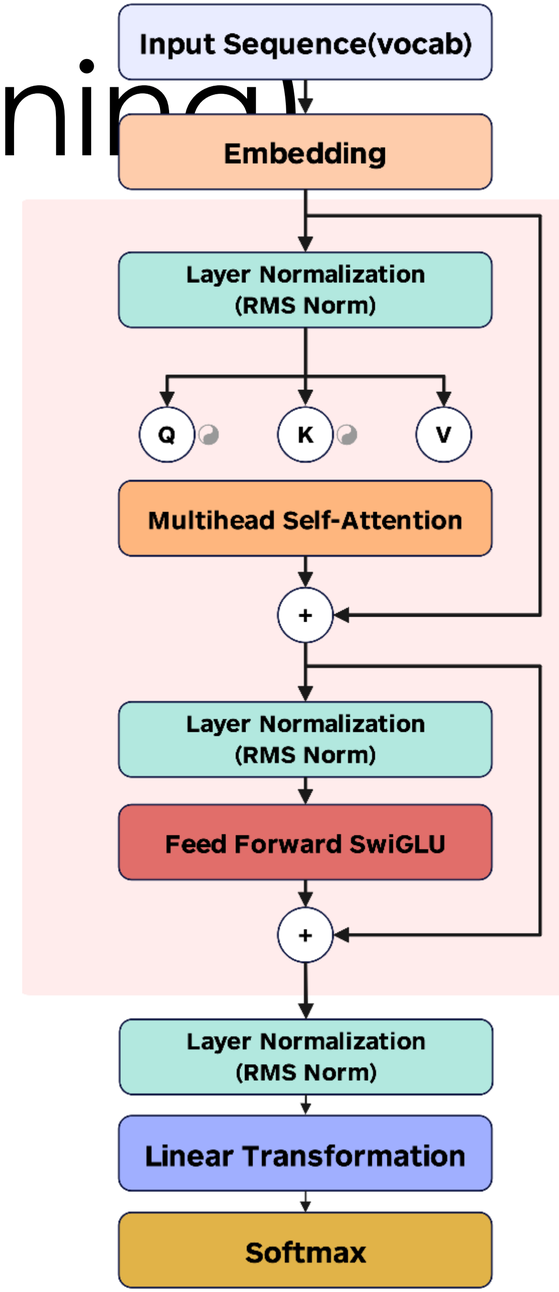
Total Flops ≈ #num_layers * (Attention block + SwiGLU block)

+ Prediction head

= #num_layers * ($6bsh^2 + 4bs^2h + 3bs^2n + 2bsh^2$)

+ #num_layers (6bshi)

+ 2 bshv



LLama 2 7B Flops Forward Calculation (Training)

Hyperparameters:

Batch size: b=1

Sequence length: s=4096

The number of attention heads: n=32

Hidden state size of one head: d=128

Hidden state size: h =4096

SwiGLU proj dim: i=11008

Vocab size: v=32000

The number of layers: N=32

$$\text{Total Flops} \approx N * (6bsh^2 + 4bs^2h + 3bs^2n + 2bsh^2)$$

$$+ N\ (6bshi)$$

$$+\ 2\ bshv$$

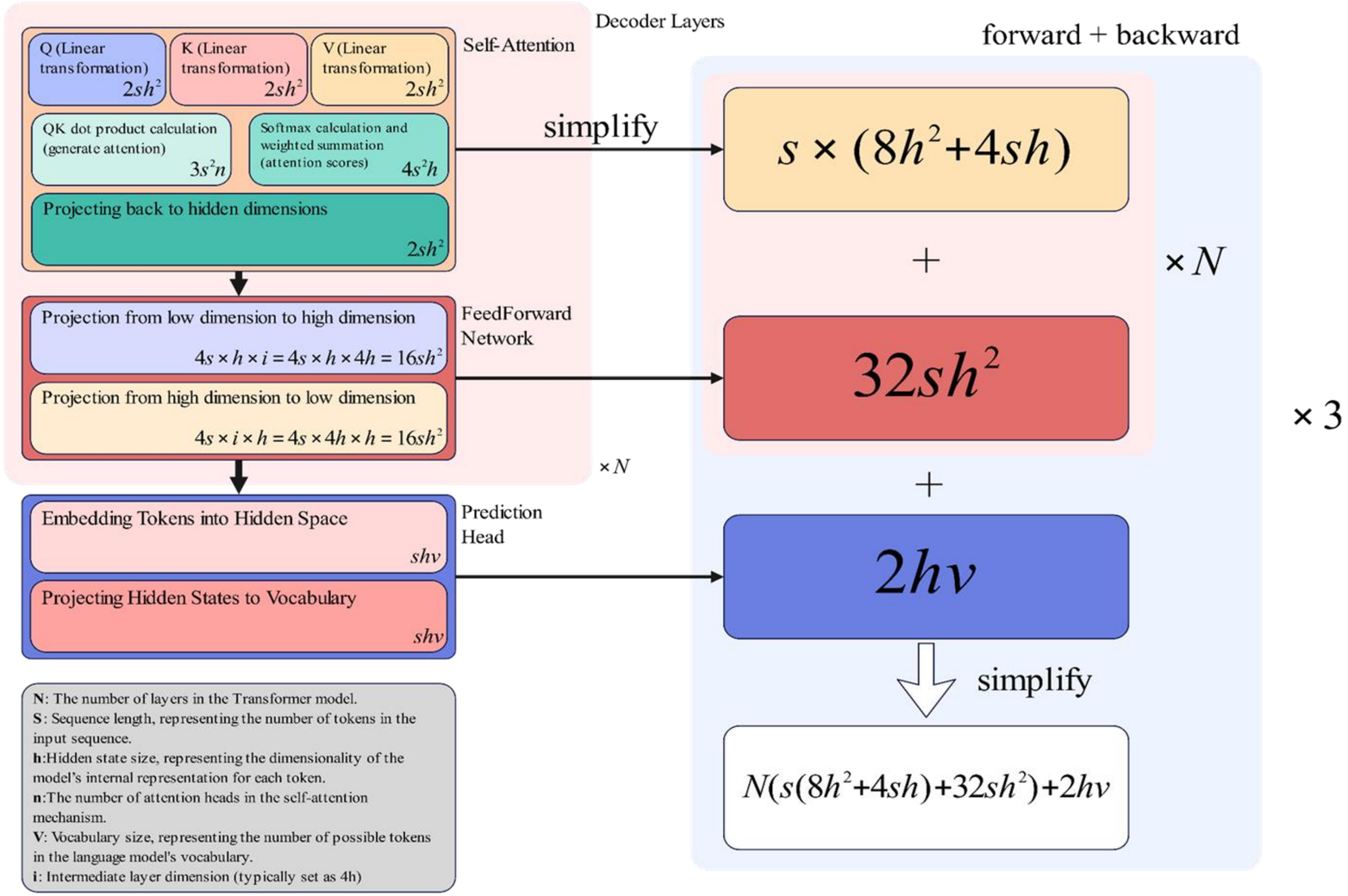
$$\approx 63\ \text{TFLOPs}$$

Flops Distribution

Training Computational Costs Breakdown:

- **Total Training TeraFLOPs:** 192.17 TFLOPs
- **FLOP Distribution by Layer:**
  - **Embedding Layer:** 1.676%
  - **Normalization:** 0.007%
  - **Residual:** 0.003%
  - **Attention:** 41.276%
  - **MLP (Multi-Layer Perceptron):** 55.361%
  - **Linear:** 1.676%

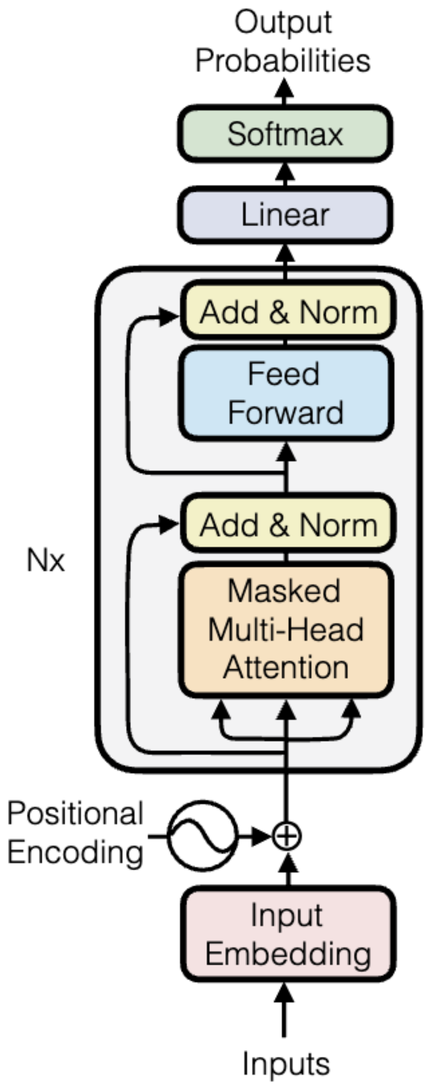
Scaling Up: Where is the Potential Bottleneck?



Connecting the Dots: Compute/Comm characteristic of LLMs

Key characteristics: compute, memory, communication

- calculate the number of parameters of an LLM?
- calculate the flops needed to train an LLM?
- calculate the memory needed to train an LLM?



After-class practice