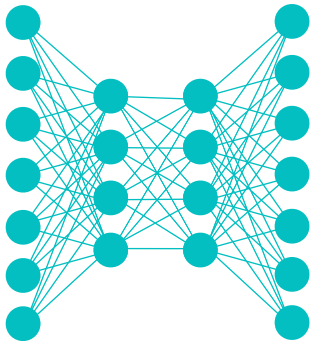
Lecture Notes for

Neural Networks and Machine Learning

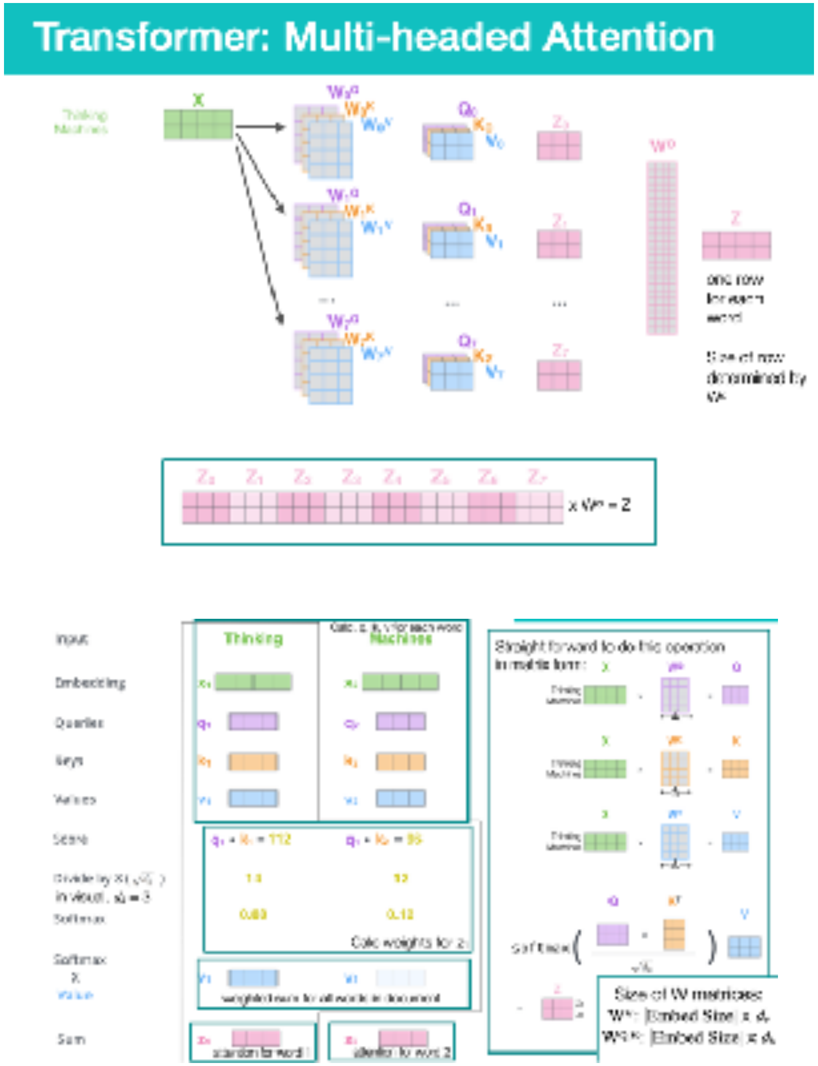
Practical Transformers

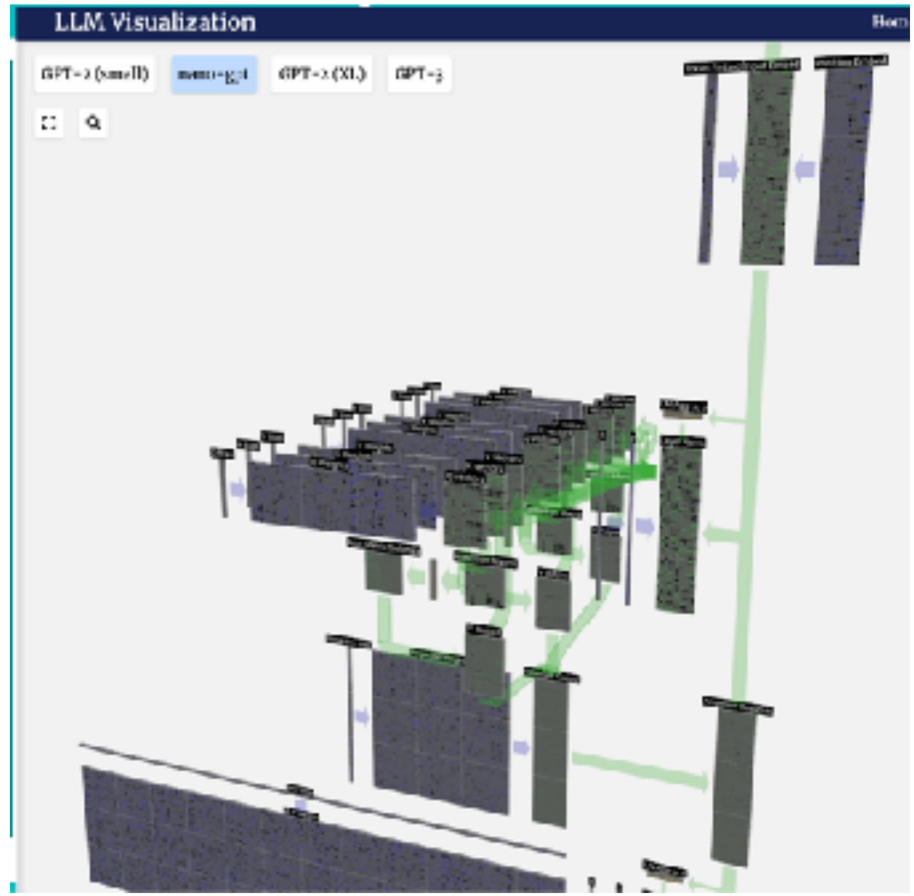


Logistics and Agenda

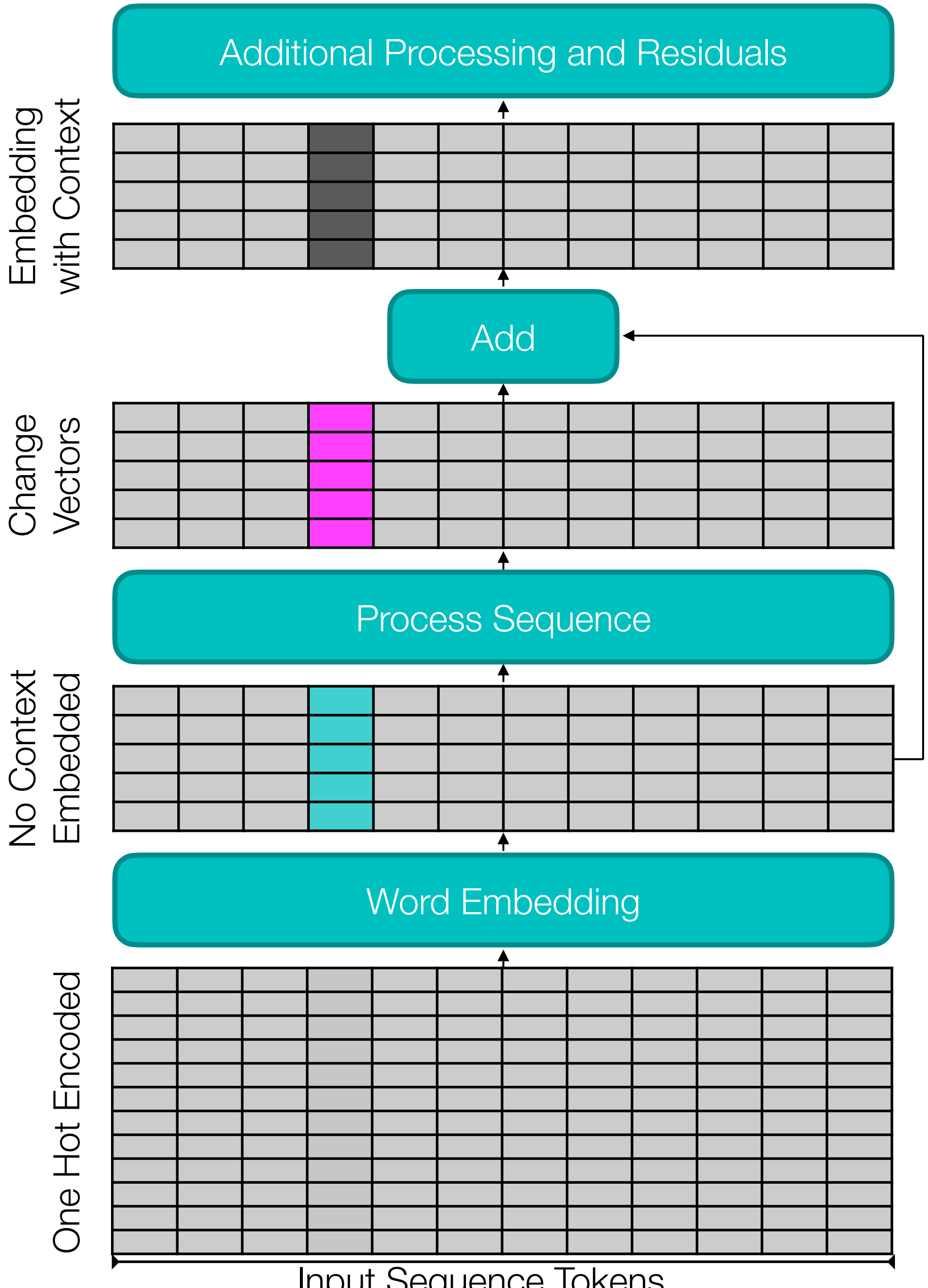
- Logistics
  - Lab due, office hours
  - Video Summary
- Agenda (probably two lectures)
  - Efficient Transformers
  - Practical Transformers
  - Student Paper Presentation

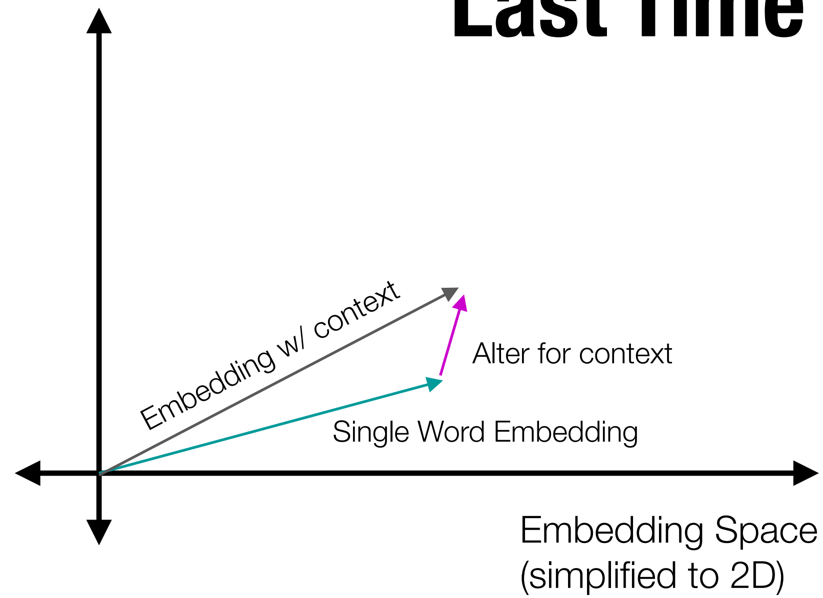
Last Time: Transformers





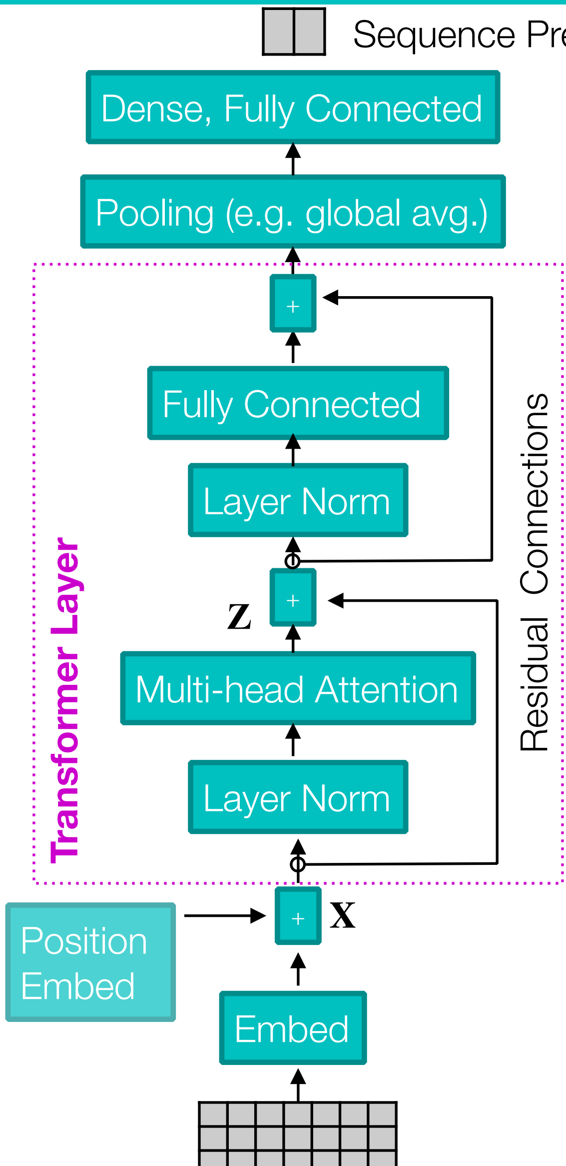
Last Time

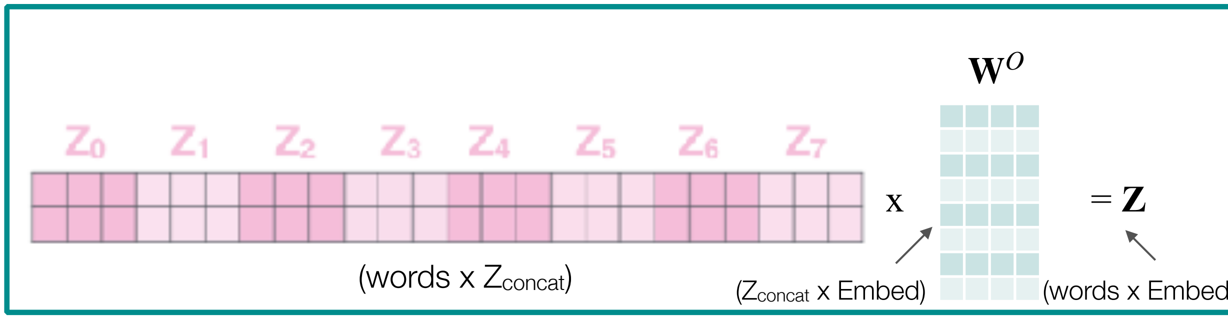




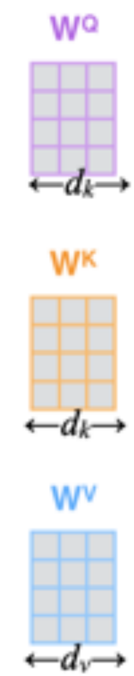
- Transformer is tasked with generating the “alter context” vector
- But this happens in a high dimensional space (word embedding space)
- Need to work in even lower dimension
- Look at all words in the sequence
- Expand back to original embedding space for addition
- Residual also helps with gradient

All together: Transformer Review

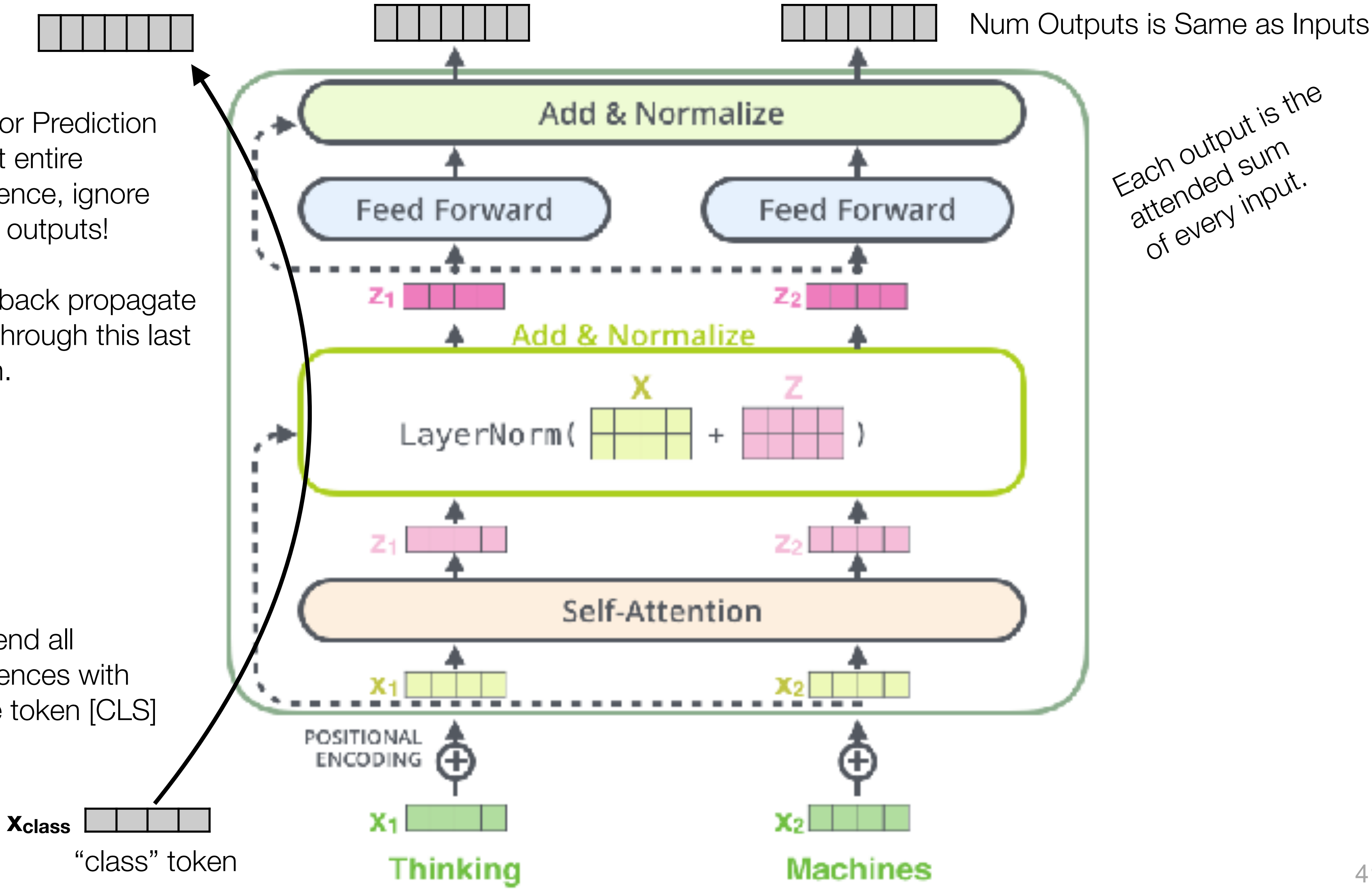





```
tf.keras.layers.MultiHeadAttention(
     num_heads,      (Number of heads Z1-Z7)
     key_dim,        (size of query/key dk)
     value_dim,      (size of each dv)
     output_shape,   (control size of Wo,
                          usually same as X)
       …
)
```



Transformer for Sequence Classification

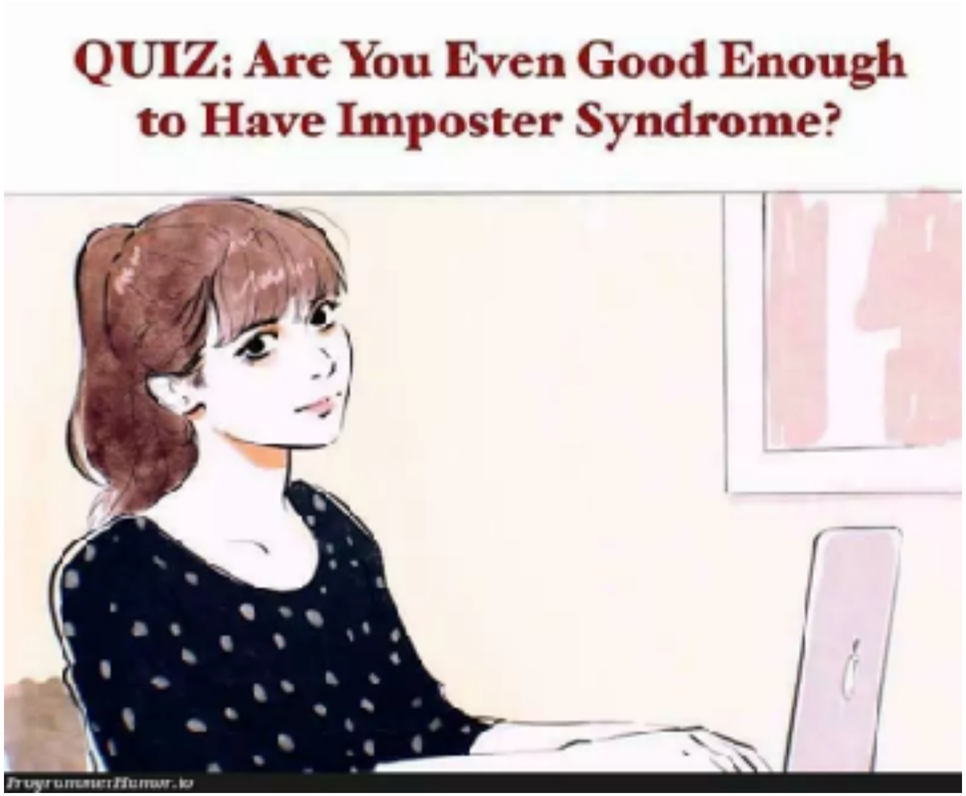


Use for Prediction about entire sequence, ignore other outputs!

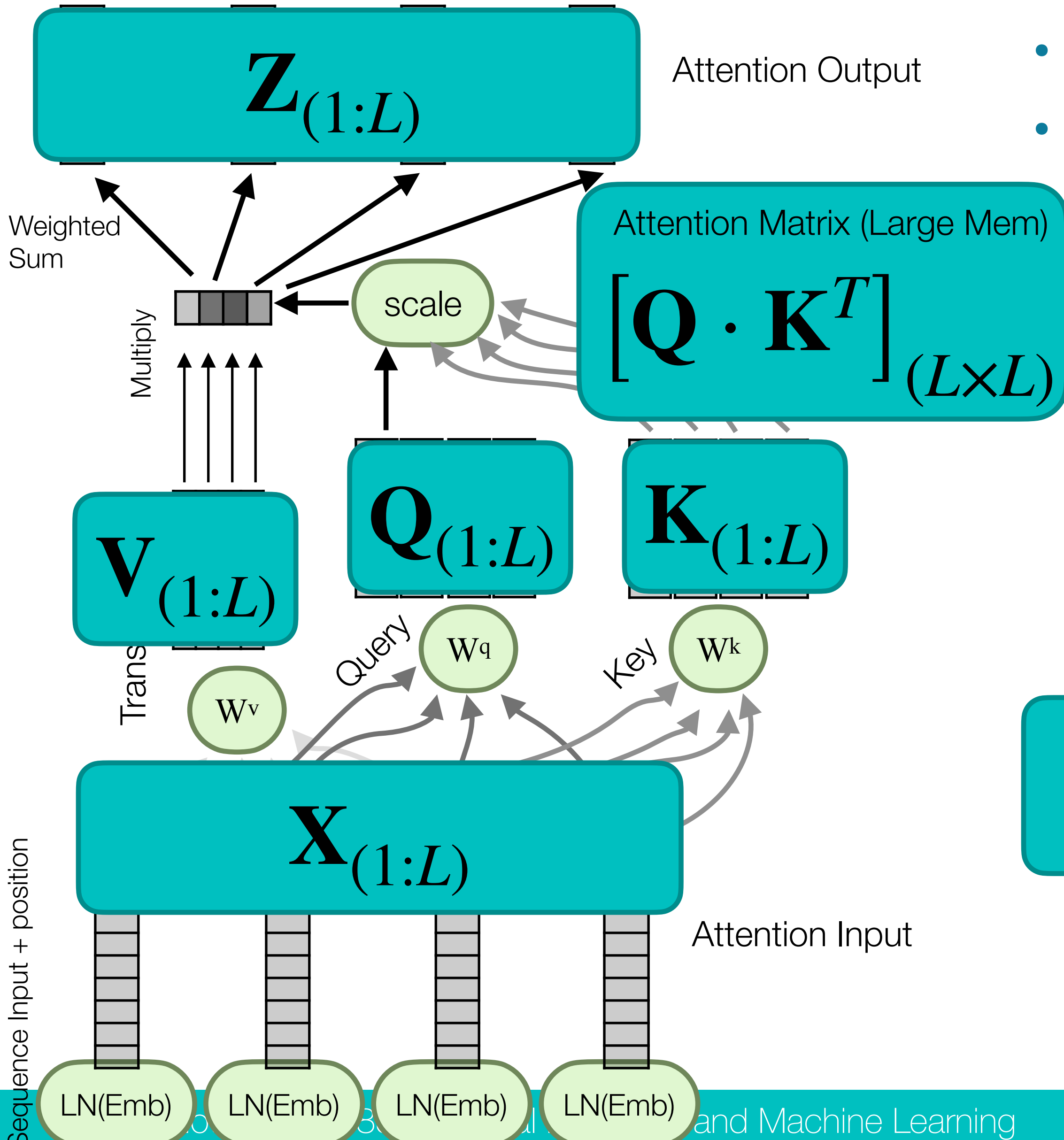
Only back propagate loss through this last token.

Prepend all sequences with same token [CLS]

Altering Self-Attention



Self Attention Overview (Review)



- Trained: $\mathbf{W}^v, \mathbf{W}^q, \mathbf{W}^k$
- Other Parameters:
  - L: length of sequence
  - Query/Key dimension, d_k
  - Value dimension, d_v
  - Type of positional encoding (more later)

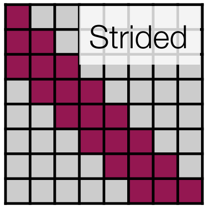
$$\text{softmax}\left(\frac{\mathbf{Q}\cdot\mathbf{K}^T}{\sqrt{d_k}}\right)\cdot\mathbf{V}$$

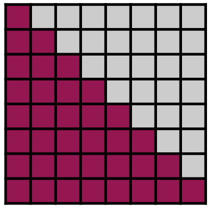
Attention Efficiently

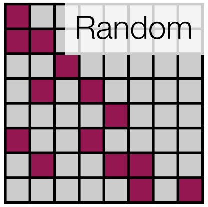
Partial Global + Rand + Strided
Zaheer et al., **BigBird**, NeurIPS 2021

Naive Implementation:
- Computation: $O(L^2 \cdot d)$
- Memory: $O(L^2 + L \cdot d)$

One idea: limit non-zero values of $\mathbf{Q} \cdot \mathbf{K}^T$
Need to define sparsity before computation







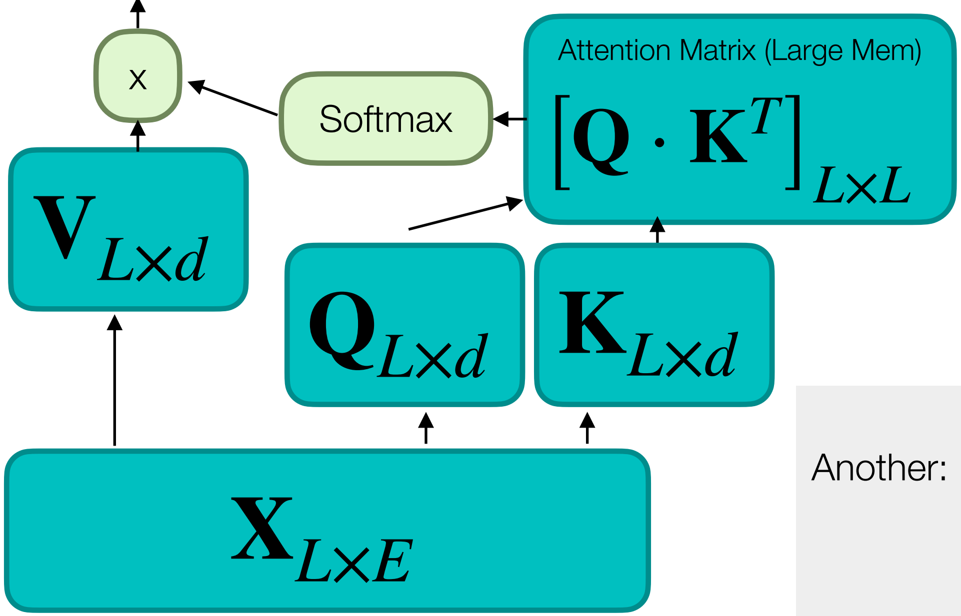
Another idea: change softmax, to allow associative rule application

$$\left(\mathbf{Q} \cdot \mathbf{K}^T\right) \cdot \mathbf{V} = \mathbf{Q} \cdot \left(\mathbf{K}^T \cdot \mathbf{V}\right)$$

$O(L^2 \cdot d)$ $\quad$ $O(L \cdot d^2)$

Efficient Implementation:
- Computation: $O(L \cdot d^2)$
- Memory: $O(d^2 + L \cdot d)$

$$\text{softmax}\left(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{d}}\right) \cdot \mathbf{V}$$



but we need a function that satisfies

$$f\left(\mathbf{Q} \cdot \mathbf{K}^T\right) = f(\mathbf{Q}) \cdot f(\mathbf{K}^T)$$

One function: softmax along rows and columns

$$\text{softmax}(\mathbf{Q}) \cdot \left(\text{softmax}(\mathbf{K})^T \cdot \mathbf{V}\right)$$

Katharopoulos et al., **Trans are RNNs**, ICLR 2021

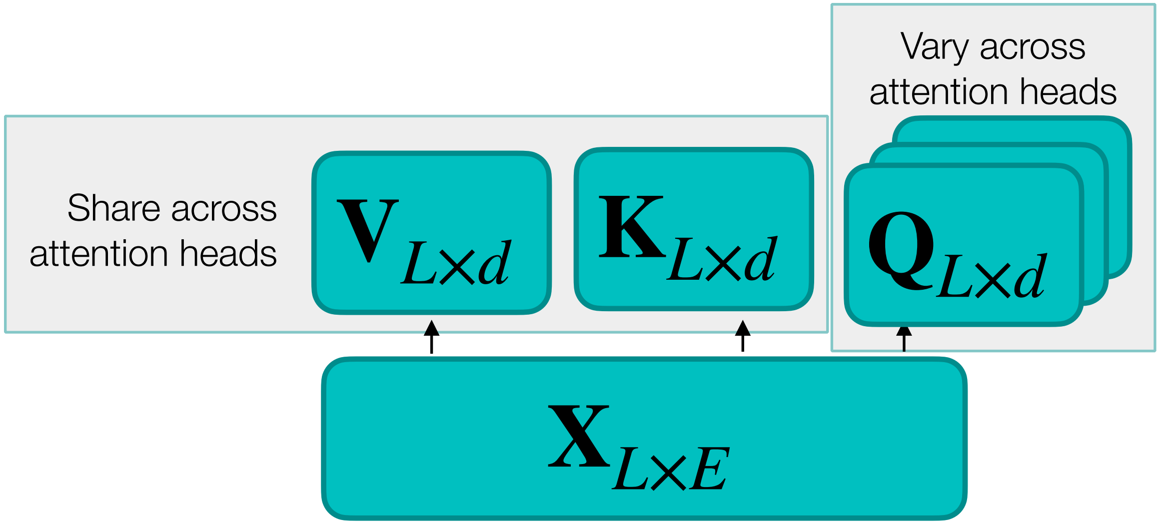
Another: $\frac{\mathbf{Q}}{\|\mathbf{Q}\|} \cdot \left(\frac{\mathbf{K}}{\|\mathbf{K}\|}^T \cdot \mathbf{V}\right) \rightarrow \frac{\mathbf{Q} \cdot \mathbf{K}^T}{\|\mathbf{Q}\|\|\mathbf{K}\|} \cdot \mathbf{V}$

same as cosine similarity

Mongaras, Dohm, and Larson, **Cottention**, CC 2025

E: token embedding size, ***L***: sequence length, ***d***: transformer dimension

Multi-query Attention (MQA)

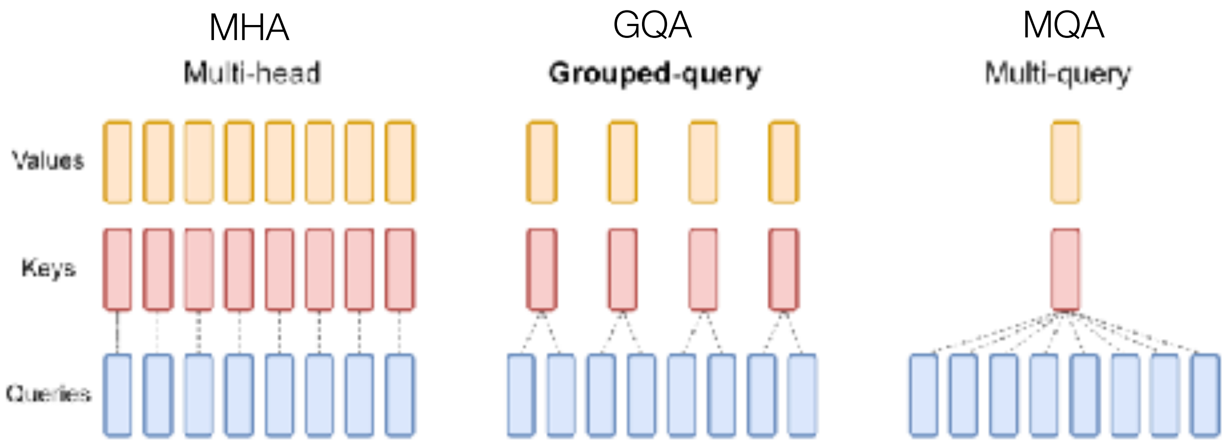


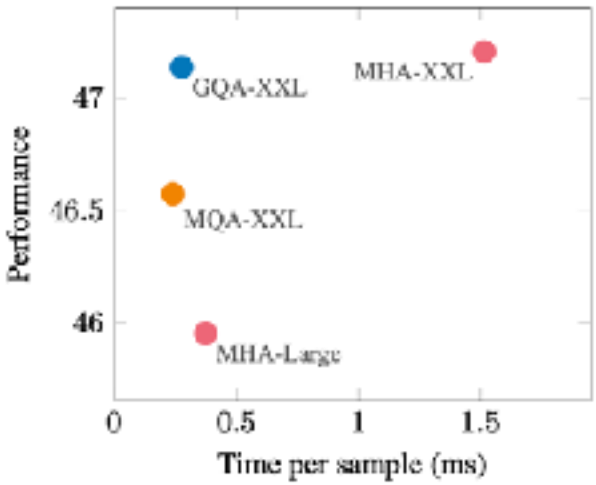
- (-) Slight drop in accuracy for various tasks
- (+) Allows larger transformer feed forward layers
- (+) larger context lengths fit in GPU memory
- (-) No speed up for distributed compute as K, V are copied

- Vanilla transformer can store V and K on SRAM of GPU, then just load in Q from high-bandwidth memory (HBM)
  - memory transfer is critical bottleneck for GPU, so you get a huge speed up
- For methods that can calculate $\mathbf{Q}_i \cdot \left(\mathbf{K}^T \cdot \mathbf{V}\right)$, the entire $\mathbf{K}^T \cdot \mathbf{V}$ matrix can be precomputed (*may not fit in SRAM for long sequences*)

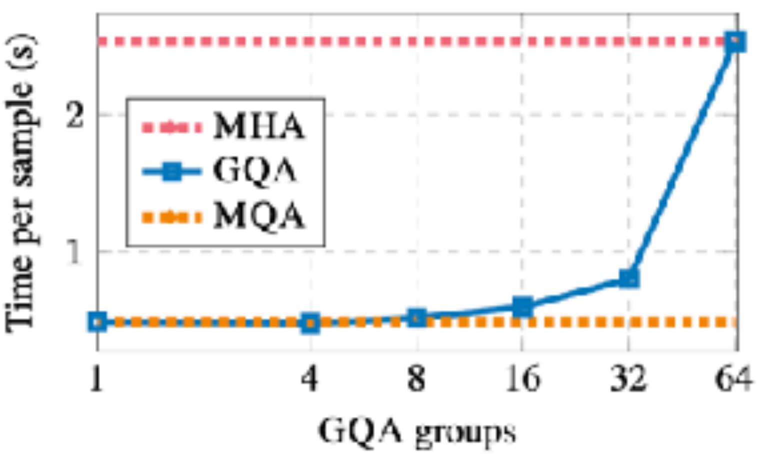
Shazeer, Fast Transformer Decoding, ArXiV (2019)

Group Query Attention





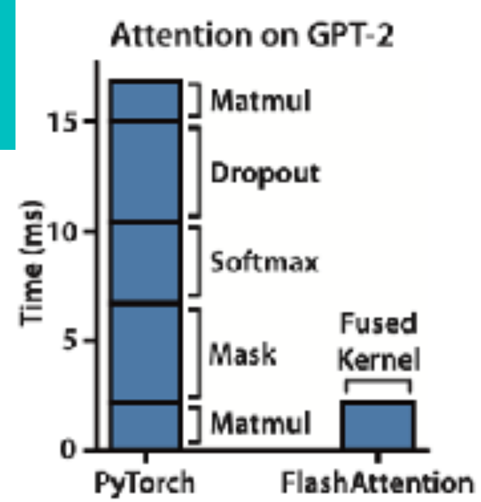
- Can take advantage of distributed computation, parallelize across groups for unique K, V
- Easy to tradeoff performance of MHA with compute of MQA

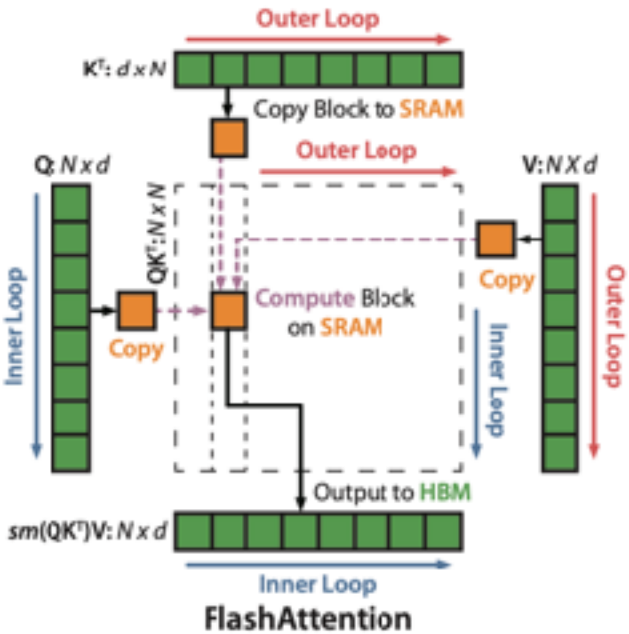


Ainslee et al., GQA, ArXiV (2023)

Flash Attention

- Calculate attention in tiles (local compute)
  - Requires calculation and saving additional variables in each tile
- During tile aggregation, scale the variables properly to get exact softmax across tiles (distributed softmax)
- Tile calculation is a shader function, massive speed up on a GPU
- Back-prop: Only save the attention output and recompute it for back propagation to save memory
  - similar to gradient checkpointing, this adds compute but saves memory
- **Flash Attention 2:**
  - Some small improvements to matrix multiplications
  - Added support for GQA (big speed ups)
  - Flash Attention becomes 9x faster than normal attention for both training and inference





Dao et al., FlashAttention, NeurIPS 2022

Distributing Softmax in Tiles

$$\mathbf{x} = [a, b, c, d]$$

$$m(\mathbf{x}) = \max\left([a, b, c, d]\right)$$

$$f(\mathbf{x}) = \left[e^{a-m(\mathbf{x})}, e^{b-m(\mathbf{x})}, e^{c-m(\mathbf{x})}, e^{d-m(\mathbf{x})}\right]$$

$$l(\mathbf{x}) = \sum f(\mathbf{x})$$

$$\text{softmax}(\mathbf{x}) = \frac{f(\mathbf{x})}{l(\mathbf{x})} = \left[\frac{e^{a-m(\mathbf{x})}}{l(\mathbf{x})}, \frac{e^{b-m(\mathbf{x})}}{l(\mathbf{x})}, \frac{e^{c-m(\mathbf{x})}}{l(\mathbf{x})}, \frac{e^{d-m(\mathbf{x})}}{l(\mathbf{x})}\right]$$

Regular Softmax Calculation

$$\mathbf{x}^{(1)} = [a, b] \quad f(\mathbf{x}^{(1)}) = \left[e^{a-m(\mathbf{x}^{(1)})}, e^{b-m(\mathbf{x}^{(1)})}\right]$$

$$\mathbf{x}^{(2)} = [c, d] \quad f(\mathbf{x}^{(2)}) = \left[e^{c-m(\mathbf{x}^{(2)})}, e^{d-m(\mathbf{x}^{(2)})}\right]$$

$$m(\mathbf{x}) = \max\left(m(\mathbf{x}^{(1)}), m(\mathbf{x}^{(2)})\right)$$

Need to track this

$$s_i = e^{m(\mathbf{x}^{(i)})-m(\mathbf{x})}$$

Slightly more FLOPS, but better utilization of parallelism

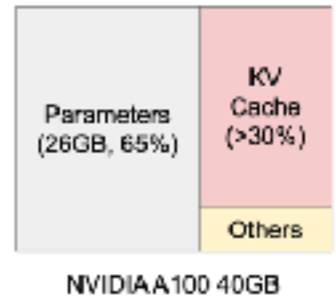
$$f(\mathbf{x}) = \left[s_1 \cdot f\left(\mathbf{x}^{(1)}\right), s_2 \cdot f\left(\mathbf{x}^{(2)}\right),\right]$$

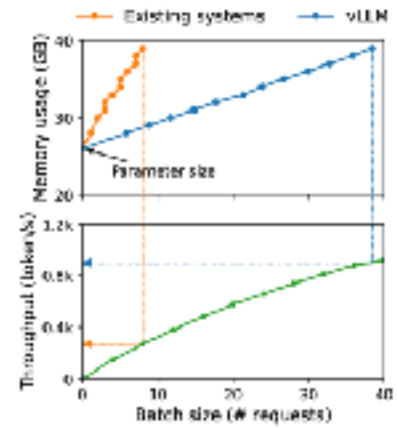
$$l(\mathbf{x}) = s_1 \cdot l\left(\mathbf{x}^{(1)}\right) + s_2 \cdot l\left(\mathbf{x}^{(2)}\right)$$

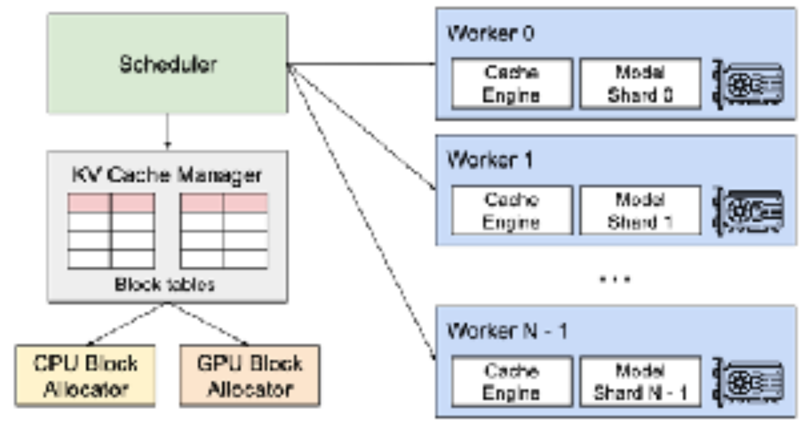
Distributed

Paged Attention

- Straight up computer science issue
- For large sequences, KV operation is fragmented across memory and compute
  - Hard to cache KV accesses
- Page KV cache in memory aligned blocks similar to virtual memory on OS
  - Vastly reduces fragmentation and speeds up accesses





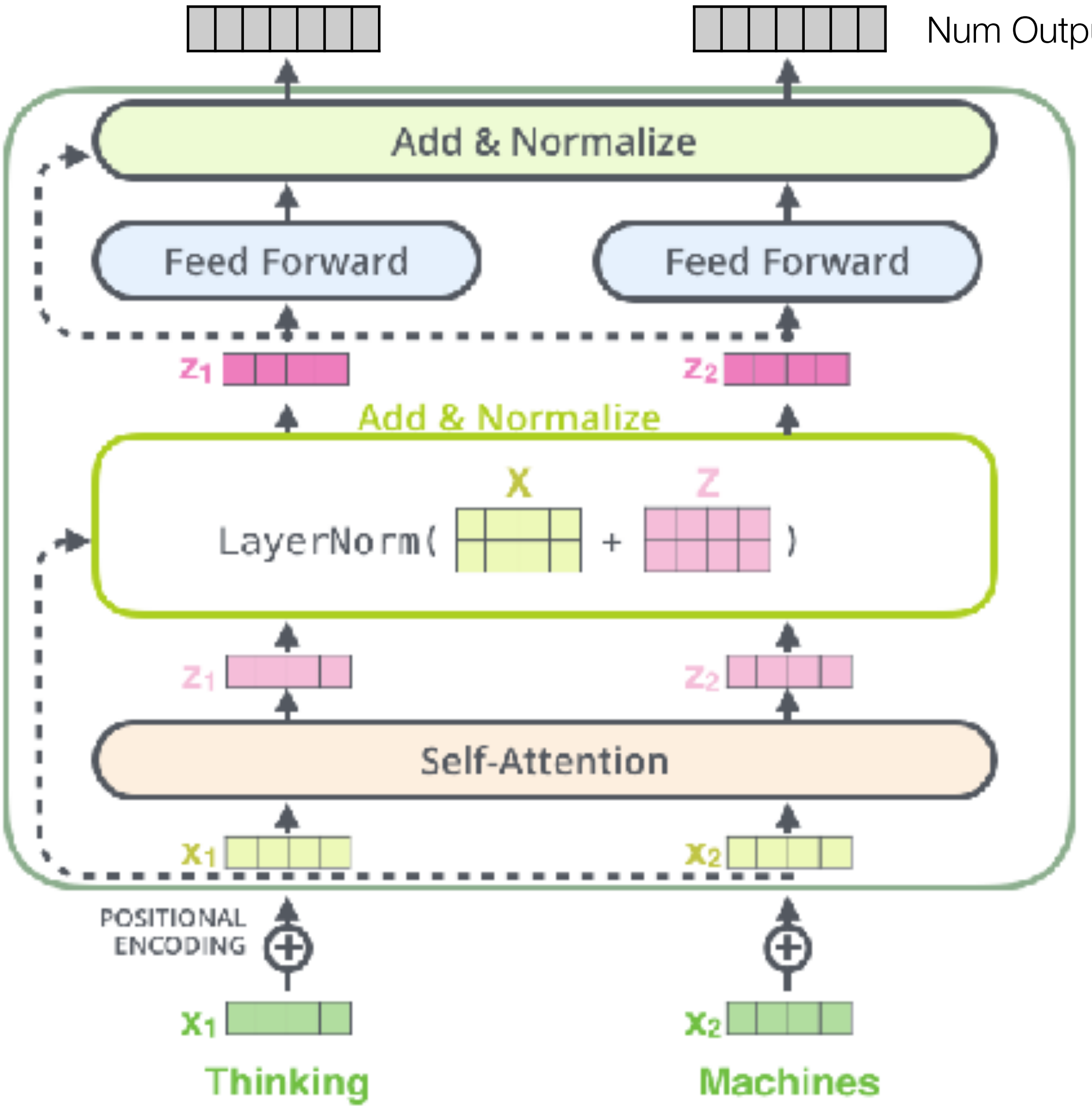


- Store the KV cache in non-contiguous blocks for better parallelized access.
- Supports parallelized sequence access with fewer misses.

Kwon et al., Efficient Memory , SOSP (2023)

Positional Encoding

Transformer for Sequence Classification



Num Outputs is Same as Inputs

Do these outputs change, if the input sequence changes order?

The order of vectors will change, but not the values of each vector…

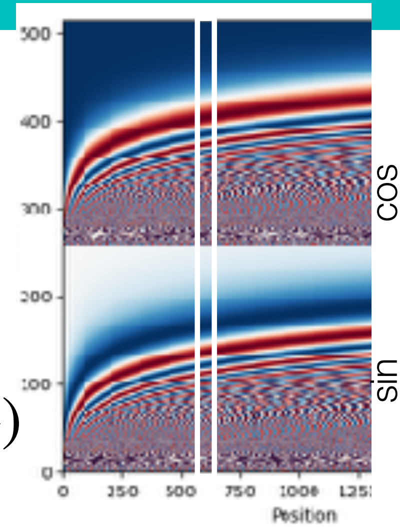
$$\text{softmax}\left(\frac{\mathbf{Q}\cdot\mathbf{K}^T}{\sqrt{d_k}}\right)\cdot\mathbf{V}$$

$\mathbf{V}_{(1:L)}$ $\mathbf{Q}_{(1:L)}$ $\mathbf{K}_{(1:L)}$

$\mathbf{X}_{(1:L)}$

Transformer: First Positional Encoding

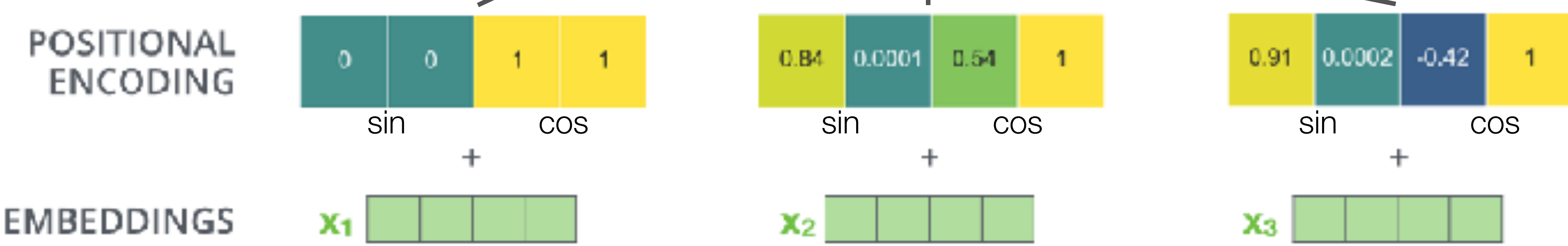
- Objective: add notion of position to embedding
- Attempt in paper: add sin/cos to embedding



p: in sequence
d_m: 1/2 dim of embed
i = index in vector

$$PE_{(p,i\in 0\ldots d_m-1)} = \sin(p/10000^{i/d_m})$$
$$PE_{(p,i\in d_m\ldots 2d_m)} = \cos(p/10000^{(i-d_m)/d_m})$$

Now use the new embeddings, with position, into transformer architecture

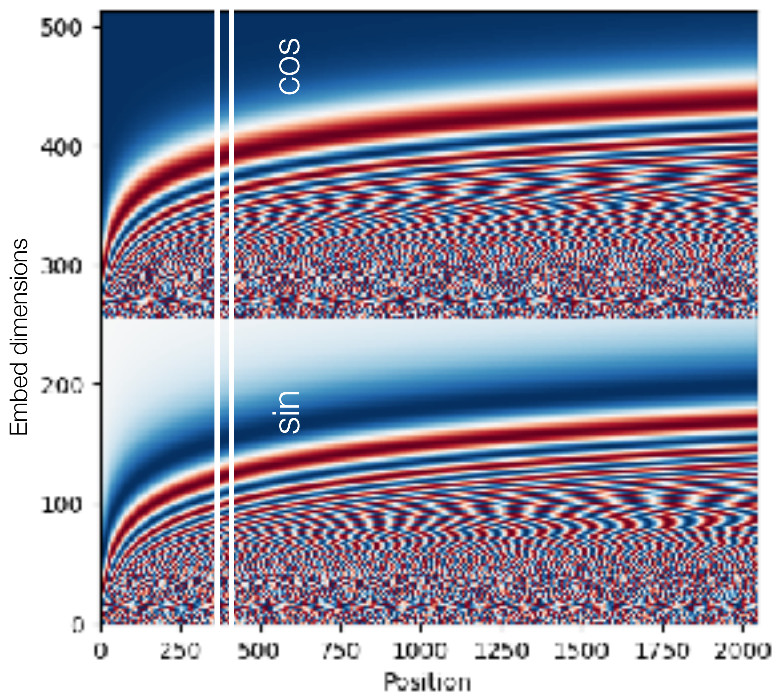


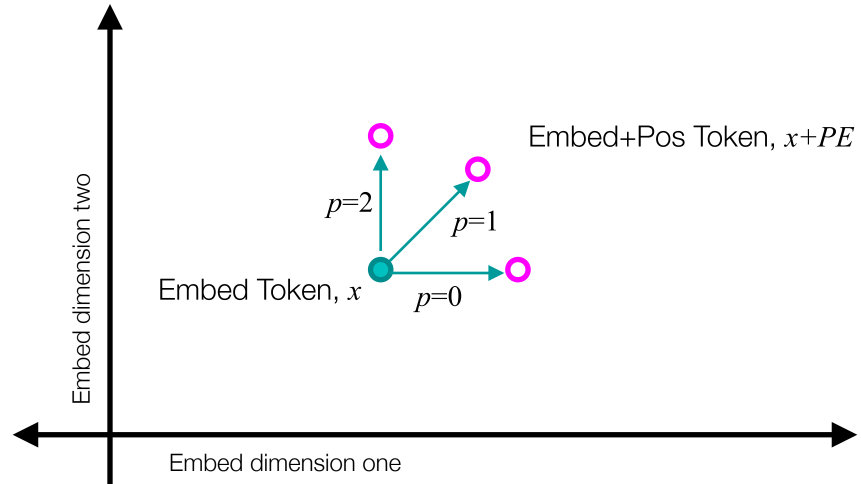
p=1, $i \rightarrow$ p=2, $i \rightarrow$ p=3, $i \rightarrow$

Hypothesis: Now the word proximity is encoded in the embedding matrix, with other pertinent information. Well, it does help… so it could be true that this is a good way to do it.

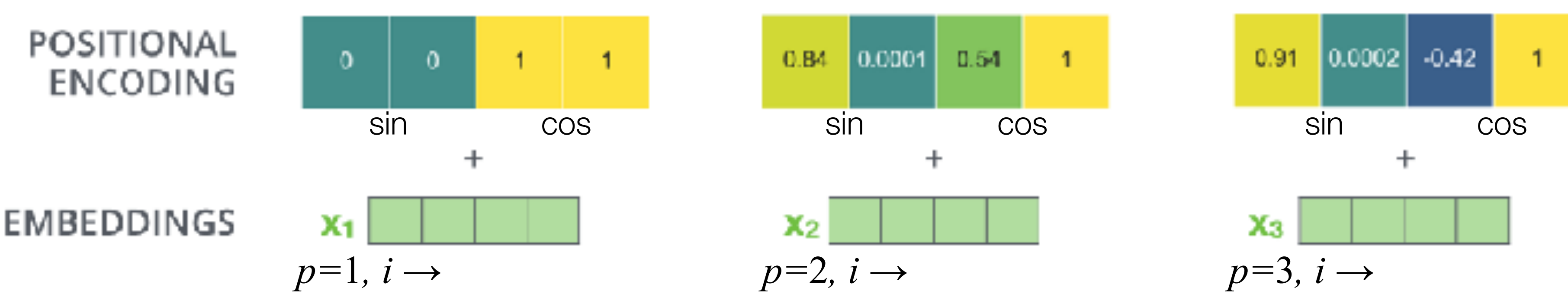
Excellent Blog on Transformers: http://jalammar.github.io/illustrated-transformer/

Positional Intuition, Geometrically



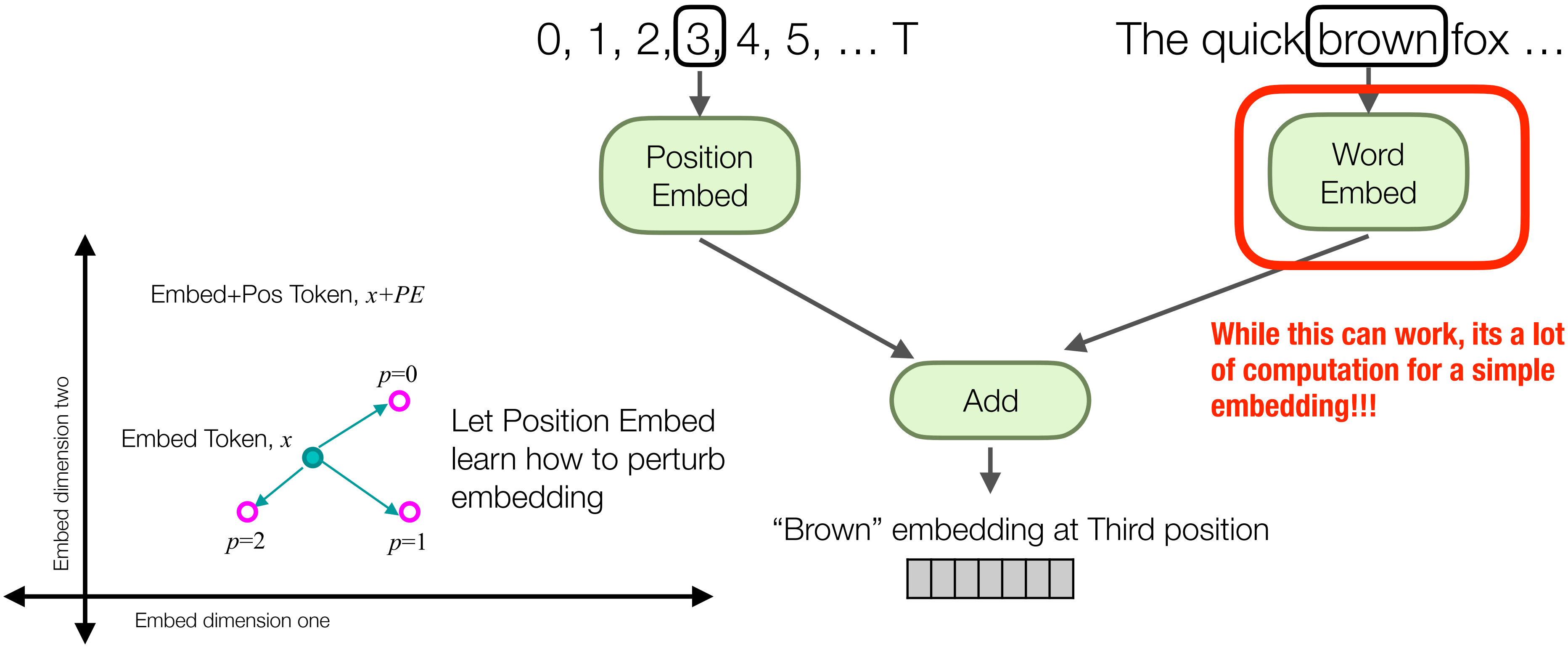


Assume that transformer model can learn the small perturbations in positional encoding, and how to use them in the context of the word.



Transformer: Positional Embedding

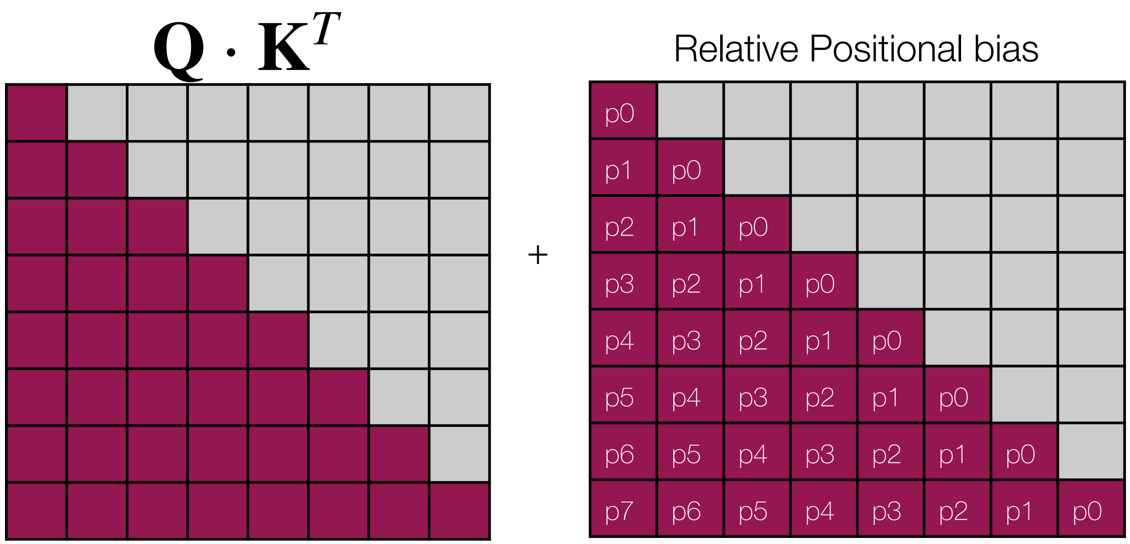
- Objective: add notion of position to embedding
- Attempt in original paper: add sin/cos to embedding
- **But could be anything that encodes position, like:**



Excellent Blog on Transformers: http://jalammar.github.io/illustrated-transformer/

Relative Positional Encoding

- Relative position encoding:
  add relative words differences into $\mathbf{Q} \cdot \mathbf{K}^T$

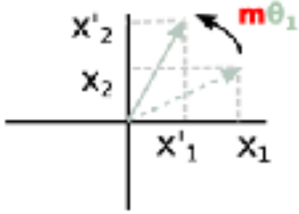


- (+) nicely structured position information
- (-) Slow, more memory
- (-) fragments ops further, more KV cache misses

- **How might we still encode relative position, without all the overhead?**

Rotary Position Embedding (RoPE)

- We rotate vectors to keep their dot product constant, even when positionally encoded!



$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} W^{(11)}_{\{q,k\}} & W^{(12)}_{\{q,k\}} \\ W^{(21)}_{\{q,k\}} & W^{(22)}_{\{q,k\}} \end{pmatrix} \begin{pmatrix} x^{(1)}_m \\ x^{(2)}_m \end{pmatrix}$$

- Except we don't do that at all, and we do not want to re-write the motivation of our paper, so here is actual RoPE:

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \boldsymbol{R}^d_{\Theta,m} \boldsymbol{W}_{\{q,k\}} \boldsymbol{x}_m$$

$$\boldsymbol{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix}$$

High frequency,
sensitive to position

Transformer learns to encode positionally sensitive meaning in high frequency indices…

Low frequency,
less sensitive to position

Fast to implement with two point wise vector multiplies and addition

In general produces better results in most tasks, not too computation