CS190C Lec5

Build Transformer Decoder

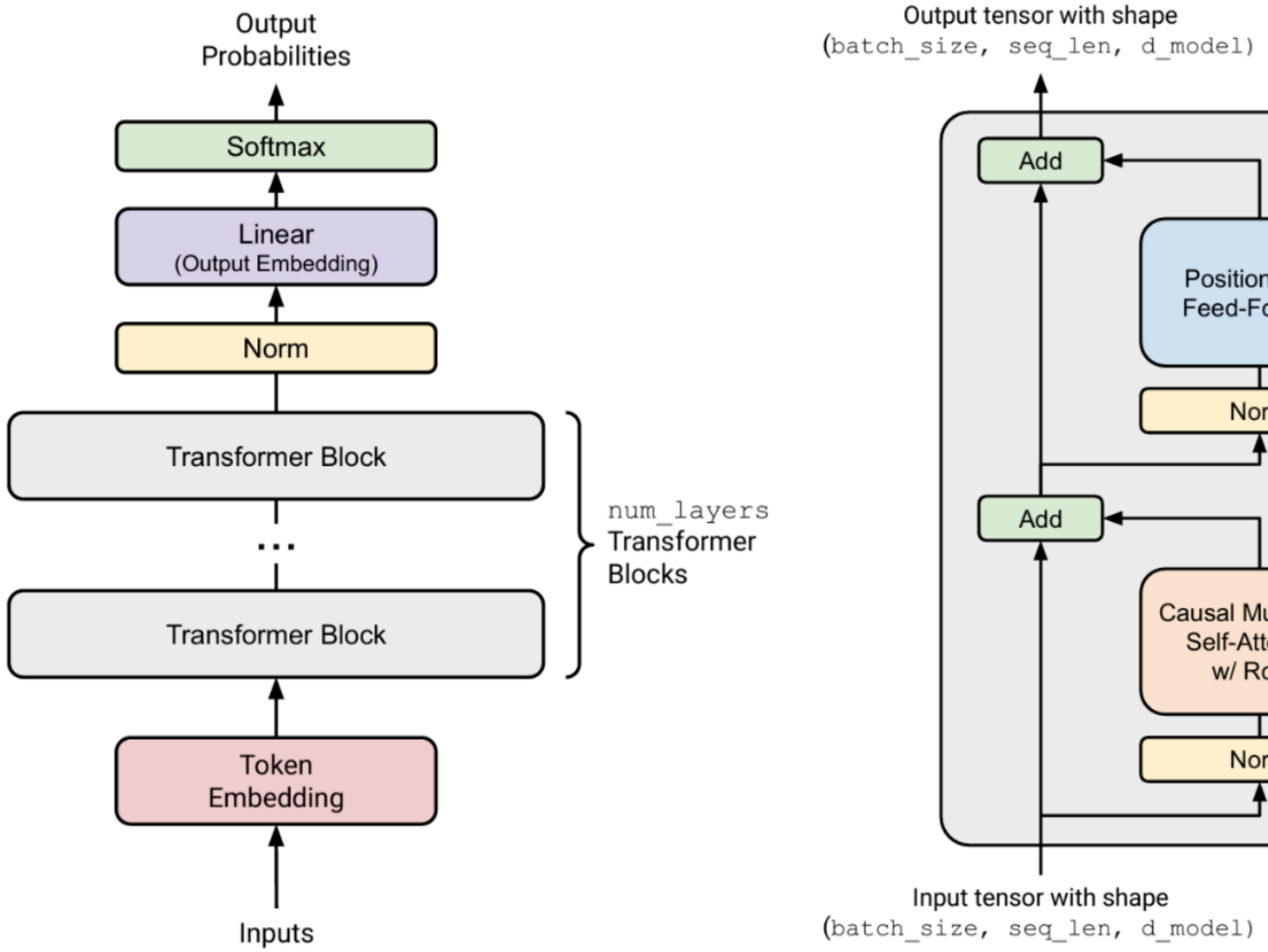
Attention please: These slides delete and only delete code implementation of all modules.

Overview

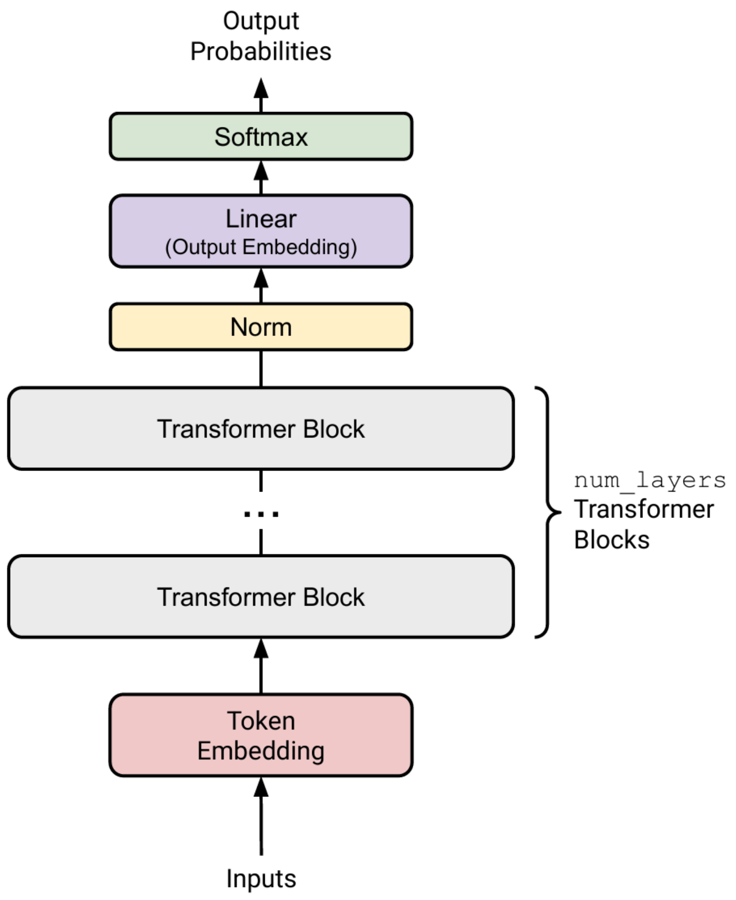
- Macro System Architecture
- Code Implementation of Basic Computational Components
- Code Implementation of RoPE
- Code Implementation of FFN and Attention
- Assembly of Transformer Block and Final Model

PART1: Macro System Architecture

Transformer Overall Architecture Diagram

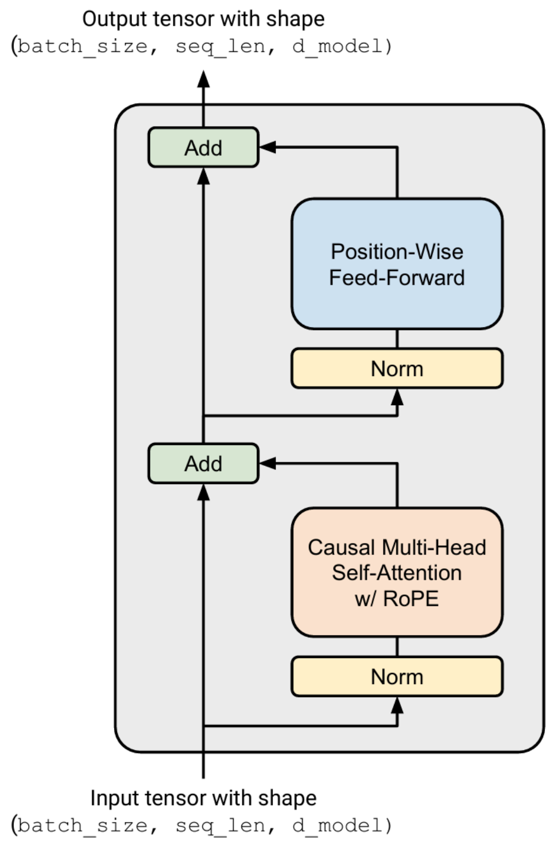


Modules of the whole Transformer



- Pass through a `Token Embedding` module to turn to dense vector (Input layer)
- Pass through several `Transformer Block` modules to absorb information in multiple rounds (Hidden layer)
- Normalize the number scale of final tensor
- Linear transform, calculate possible scores for generating of each word (Output layer)

Modules of each Transformer Block

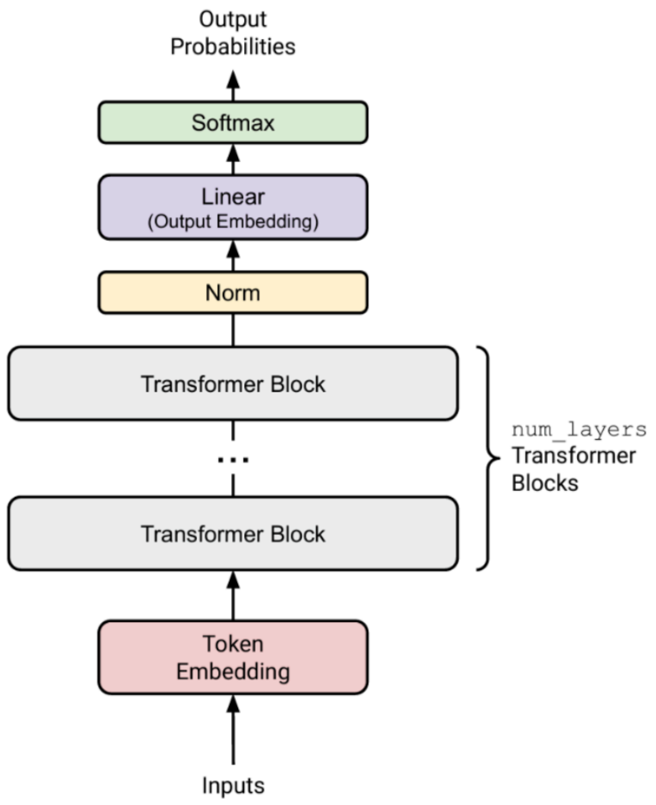


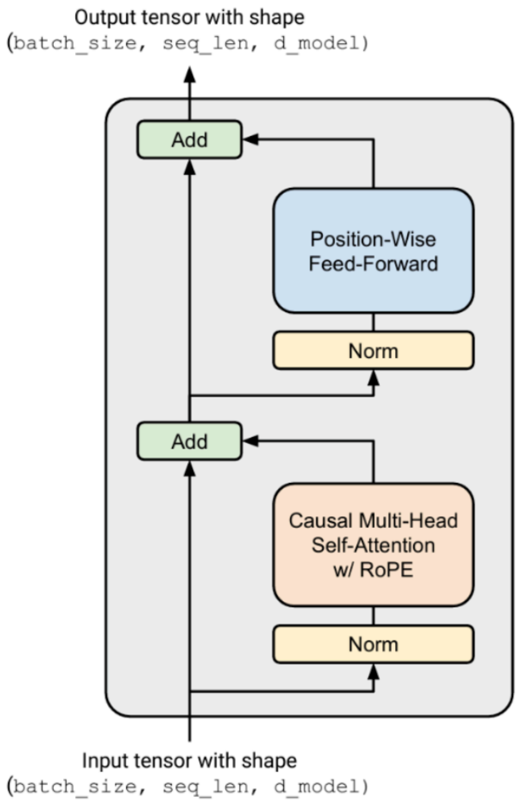
- Pre-layer RMSNorm module
- MHA Module with Residual Connection
  - contains RoPE module if position embedding needed
  - contains Softmax module for calculation of attention score
- Pre-layer RMSNorm module
- FFN Module with Residual Connection
  - contains SiLU module

Moreover, almost all modules require calling the **linear transformation** module!

PART2: Code Implementation of Basic Computational Components

Basic Components to be Implemented





Modules to be implemented:

- `Token Embedding`
- `Linear`
- `RMSNorm`
- `SiLU`
- `Softmax`

1. `class Generate_Embeddings`

IDEA:

- Most original input: BPE encoding results (e.g., `[3,10,2,6,4]`, all token IDs from the vocabulary)
- Tensor shape: `[batch_size, seq_len]`
- Expected model input: Different words in the vocabulary have different embedding vectors $\mathrm{emb}_i \in R^{d\_model}$
- Implementation idea:
  - Generate a learnable matrix $W_e \in R^{|V|*d\_model}$.
  - The i-th column represents emb_i
  - Randomly initialized W_e, representing no prior knowledge about the meaning of any word at first.

1. `class Generate_Embeddings`

Parameter scheme:

- Initialization phase: Pass in `vocab_size`, `d_model`, `device` (the device where PyTorch tensors are stored), and `dtype` (numerical type of tensor elements).
- Forward phase:
  - Pass in `token_ids` (shape `[batch_size, seq_len]`)
  - Output shape is `[batch_size, seq_len, d_model]`

1. `class Generate_Embeddings`

```
Code here
```

2. `class Linear_Transform`

IDEA:

- Assume we need to transform a 3-dimensional tensor into a 6-dimensional one...
- Mathematically speaking: Let the 3D tensor `x` $(1 * 3)$ right-multiply a $3 * 6$ matrix `W`
- The shape of `xW` is then $1 * 6$, just like the diagram below.

Question: Linear operations are the most frequent in LLMs......

Can this operation be accelerated as much as possible?

2. `class Linear_Transform`

- PyTorch tensors follow the "last dimension elements have contiguous memory addresses" principle.
- For a $3*6$ tensor W, its memory layout as the diagram(same color means contiguous):

- For a $4*3*6$ tensor (`[batch size, rows, columns]`), every 6 elements are also contiguous (e.g., the memory address difference between `[1,1,1]` and `[1,1,2]` is 1 units, and the memory address difference between `[1,1,1]` and `[1,2,1]` is 6 units).

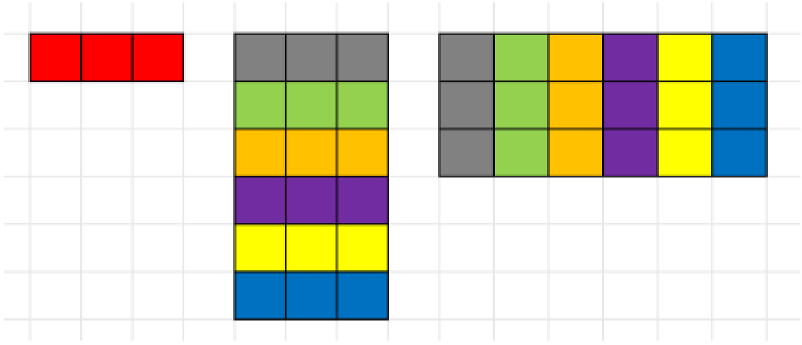
2. `class Linear_Transform`

Perform 6 vector dot product operations. In each operation:

- `x` participates as a whole row, memory is contiguous, can fully utilize cache
- `W` participates as a whole column, memory is not contiguous, may not utilize cache effectively

How can we make `W` also participate "as a whole row" in each operation?

2. `class Linear_Transform`



- PyTorch's transpose operation does not change the tensor's underlying memory space, just change stepsize. (Can you give an example?)
- Create new `W`: `[6,3]` (every 3 elements are contiguous in memory)
- Transpose `W` to `[3,6]`: Can perform matrix multiplication, and the memory distribution remains unchanged
- `x` and `W` have fully contiguous memory access during each operation, allowing full utilization of cache!

2. `class Linear_Transform`

```
Code here
```

3. `class RMSNorm`

IDEA: Normalize the input tensor $\vec{a}$ (We've discussed why at `Lec3`)

- $a_i = \frac{a_i}{RMS(\vec{a})} g_i$ (divide by normalization weight uniformly, and apply learnable fine-tuning)
- g_i is a learnable parameter
- $RMS(\vec{a}) = \sqrt{(\frac{1}{d_{model}} \sum a_i^2) + \epsilon}$, that is L2-Norm.

Input tensor shape: `[batch_size, seq_len, d_model]` $\Rightarrow$ Not a 1D vector, how to handle?

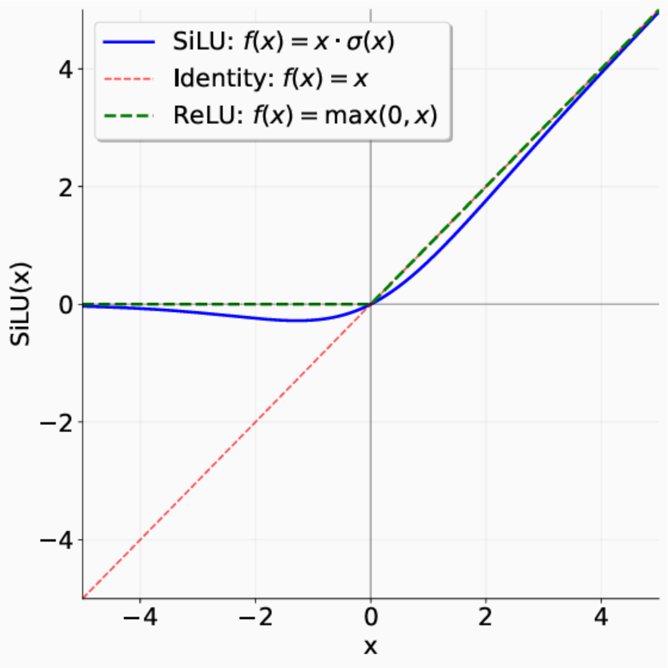
PyTorch's broadcasting mechanism:

- Operations are performed on the last few dimensions by default, previous dimensions are all replication operations

3. `class RMSNorm`

```
Code here
```

4. `class SiLU_Activation`



$$\text{SiLU:} f(x) = x \cdot \sigma(x)$$
$$\text{ReLU:} f(x) = \max(0, x)$$

- For $x < 0$: roughly equals to 0
- For $x > 0$: roughly stays the same
- Compare with ReLU: smooth and differentiable at $x = 0$

4. `class SiLU_Activation`

```
Code here
```

5. `class Softmax_Activation`

How to turn a score tensor to a distribution?

$$x_i = \frac{e^{x_i}}{\sum e^{x_i}}$$

- Each x_i calculates its exponential as a weight, then performs weight normalization
- Can make the advantage of relatively larger values more pronounced
- Even smaller values remain non-zero after normalization

Problem: What if there exists a very large x_i? (e.g., normalizing `[20,3,1005]`)

- e^{1000} = `NAN`

5. `class Softmax_Activation`

- Normalizing `[100,101,102]` vs Normalizing `[-2,-1,0]`
- Weight of 102: $\frac{e^{102}}{e^{102}+e^{101}+e^{100}}$ = $\frac{e^{0}}{e^{0}+e^{-1}+e^{-2}}$
- The Softmax normalization result of `[100,101,102]` is equivalent to that of `[-2,-1,0]`

Let x_{max} be the maximum value among x_i:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum e^{x_i}}$$
$$= \frac{e^{x_i}/e^{x_{max}}}{\sum e^{x_i}/e^{x_{max}}}$$
$$= \frac{e^{x_i - x_{max}}}{\sum e^{x_i - x_{max}}}$$

That is: subtract x_{max} from all x_i to avoid problems with extremely large values that cannot be calculated!

5. `class Softmax_Activation`

```
Code here
```

PART3: Code Implementation of RoPE

Review: RoPE Calculation Rules

$\vec{x} \Rightarrow R_i \vec{x}$ ($\vec{x} \in R^d$,d is even)

Divide the d-dimensional vector into several sub-segments of length 2, resulting in a total of $d/2$ sub-segments, each sub-segment makes a rotation of angle $\theta_{i,k}$. (Proportional to position i)

$$R^i = \begin{bmatrix} R_1^i & 0 & 0 & \cdots & 0 \\ 0 & R_2^i & 0 & \cdots & 0 \\ 0 & 0 & R_3^i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & R_{d/2}^i \end{bmatrix} \quad R_k^i = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}.$$

where $\theta_{i,k} = \frac{i}{\Theta^{2k/d}}$

Review: RoPE Calculation Rules

$$R^i = \begin{bmatrix} R_1^i & 0 & 0 & \cdots & 0 \\ 0 & R_2^i & 0 & \cdots & 0 \\ 0 & 0 & R_3^i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & R_{d/2}^i \end{bmatrix} \qquad R_k^i = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}.$$

Property of matrix $R : (R^m)^T R^n = R^{n-m}$

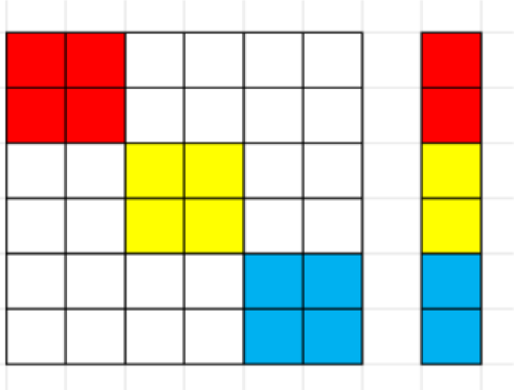
In the attention mechanism, for the query vector q_i at position i and the key vector k_j at position j :

- $q_i' = R^i q_i,\ k_j' = R^j k_j$
- $q_i'^T k_j' = q_i^T (R^i)^T R^j k_j = q_i^T R^{j-i} k_j$

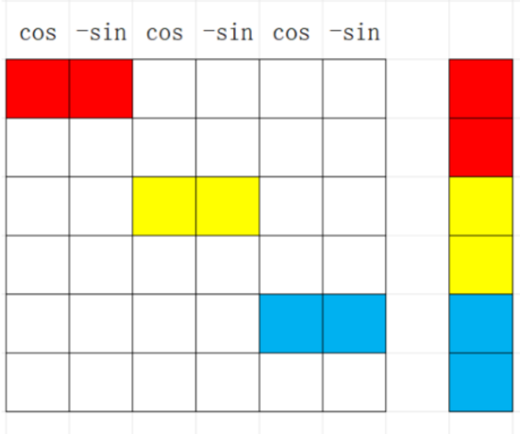
How to Avoid Brute-force Matrix Multiplication?

- Observation: Even rows of the matrix correspond to the same transformation rule: `[cos, -sin]`, with angle $\theta_{i,k}$ as the variable
- If these even rows sharing the same rule can be computed efficiently in a unified manner, how should subsequent processing proceed?

$$R^i = \begin{bmatrix} R_1^i & 0 & 0 & \cdots & 0 \\ 0 & R_2^i & 0 & \cdots & 0 \\ 0 & 0 & R_3^i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & R_{d/2}^i \end{bmatrix}$$

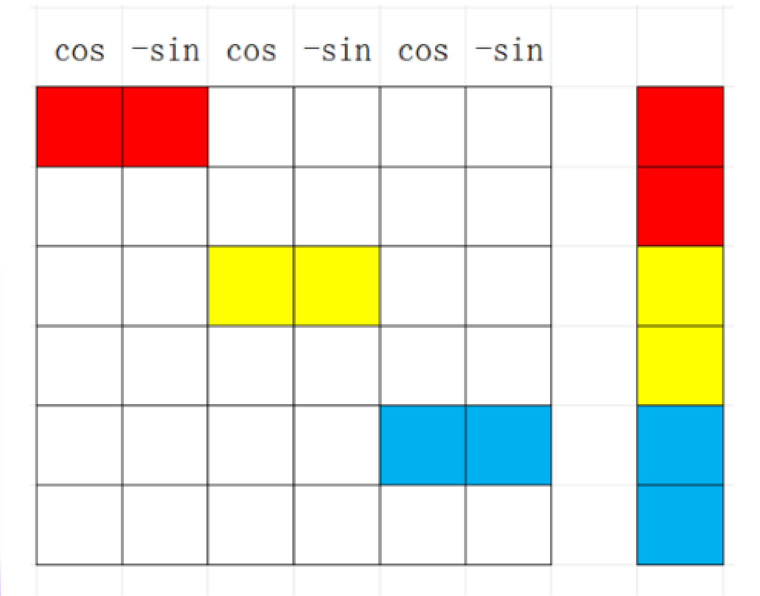


$$R_k^i = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}.$$



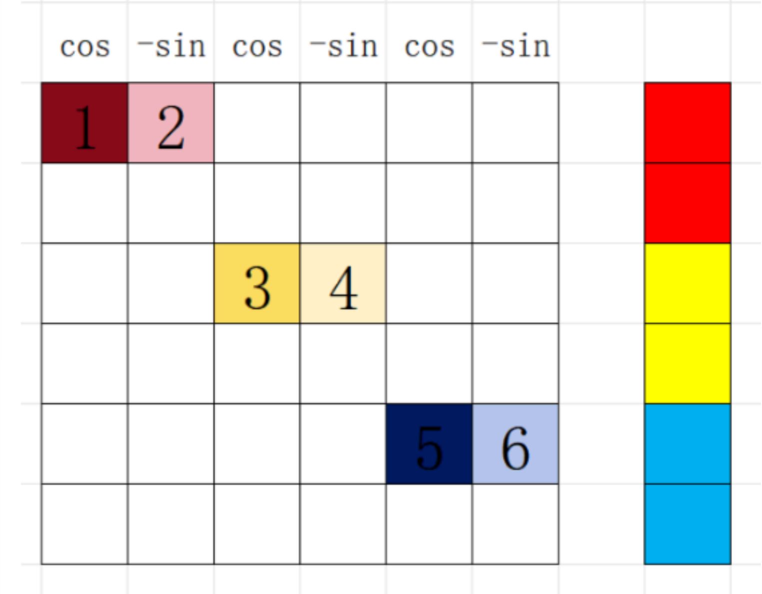
How to Avoid Brute-force Matrix Multiplication?

- Calculate products of `Red-Red`, `Yellow-Yellow`, `Blue-Blue`, which coreespond to:x_0, x_2, x_4 after RoPE
- x_1, x_3, x_5 are the same.
- Calculate each 2-number blocks, and calculate products.



How to Calculate 2-number blocks of R_i?

- Pre-calculate a $\theta_{i,k}$ lookup table of size `[max_seq_len, d/2]`
- Then calculate lookup tables of the same size for $cos(\theta_{i,k})$ and $sin(\theta_{i,k})$
- Obtain values at positions 1, 3, 5 directly from the cos lookup table
- Obtain values at positions 2, 4, 6 directly from the sin lookup table



How to Calculate 2-number blocks of R_i?

- Concatenate pairs 12, 34, 56 into 3 blocks, perform block-wise dot product with the 3 blocks of `x`, obtaining the values for the three even rows.
- Similarly, obtain the values for all odd rows of the transformed `x`, concatenate both to get the complete transformed `x` vector.

Code Implementation of RoPE

```
Code here
```

Code Implementation of RoPE

```
Code here
```

PART4: Code Implementation of FFN and Attention

1. `class Feed_Forward_Network`

Module algorithm (excluding residual connection):

- Input tensor `x` (`d_model` dimensional)
- Route 1: Transform via `W1` to `d_ff` dimensional, pass through `SiLU` activation function to get new `x`
- Route 2: Transform via `W3` for another expansion, getting gated `d_ff` dimensional
- Element-wise multiplication of `x` and gated, getting new `x`
- Transform back to `d_model` dimensional via `W2`

We call it `SwiGLU`: $\mathrm{FFN}(x) = \mathrm{SwiGLU}(x, W_1, W_2, W_3)$

1. `class Feed_Forward_Network`

```
Code here
```

1. `class Feed_Forward_Network`

```
Code here
```

2. `class Multihead_Attention`

Module algorithm (excluding residual connection):

- Input tensor `x` with size `[batch_size, seq_len, d_model]`
- Through three linear transformations, essentially linearly transforming each d_model dimensional word vector of x to get:
  - `Q`, `K` matrices `[batch_size, seq_len, num_heads*d_k]`
  - `V` matrix `[batch_size, seq_len, num_heads*d_v]`
- Perform `RoPE` positional encoding on `Q`, `K` matrices
- Generate attention upper triangular mask (to prevent breaking the "autoregressive" assumption)
- Use QKV matrices and attention mask to compute attention output

2. `class Multihead_Attention`

Method for calculating attention output:

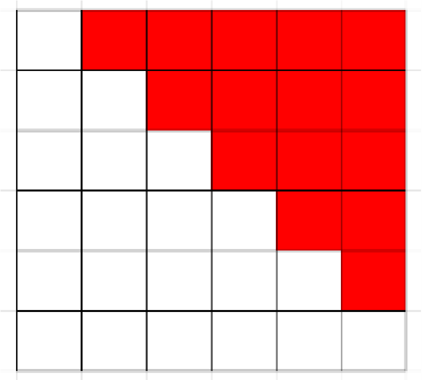
- Multiply `Q`, `K` matrices to calculate token feature matching scores
- Apply mask processing to the matching score matrix
- $\mathrm{Softmax}(\frac{QK^T}{\sqrt{d_k}})$, calculate attention scores between tokens
- Multiply the score matrix by the `V` matrix to get the attention output of each head
- Multiply `W_O` matrix to integrate heads

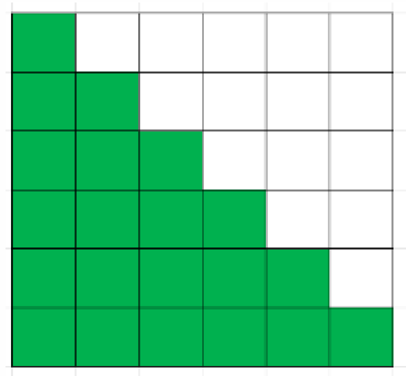
Calculation of Attention Output for One Head

```
Code here
```

Generation of Attention Mask

```
Code here
```





List of Sub-modules in Multihead_Attention

- Attention mask generation module
- Attention output calculation module (sdpa)
- RoPE module $\Rightarrow$ Requires additional parameters like max_seq_len, theta, token_positions, etc.
- Four types of linear transformations: Q, K, V, O

Assembly of Complete Module

```
Code here
```

Assembly of Complete Module

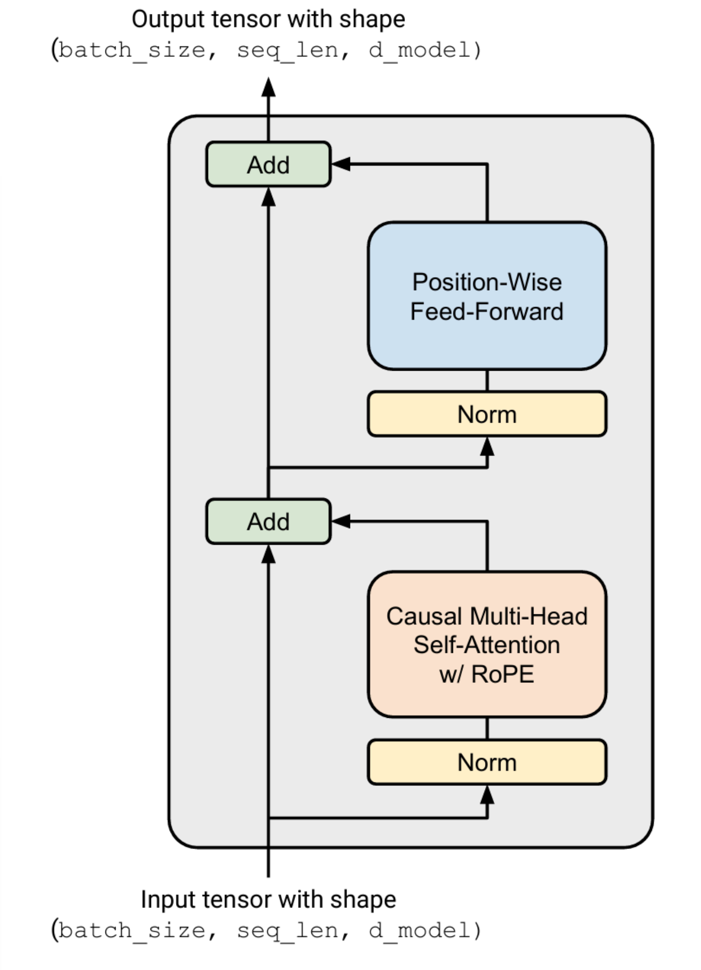
```
Code here
```

Assembly of Complete Module

```
Code here
```

PART5: Assembly of Transformer Block and Final Model

Structure of Transformer Block



- A RMSNorm module
- A MHA module
- Residual connection
- An other RMSNorm module
- A FFN module
- Residual connection

The parameters received by each Block are the union of all parameters required by the above modules!

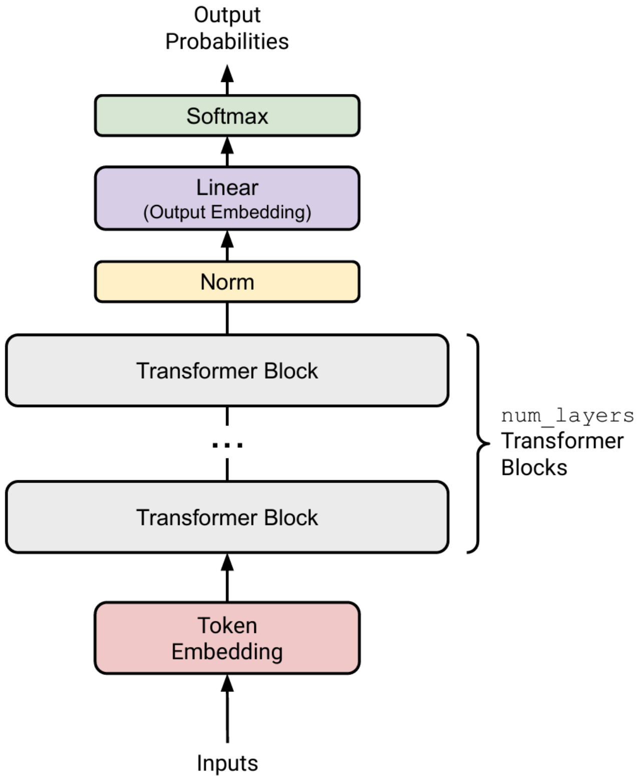
Assembly of Transformer Block

```
Code here
```

Assembly of Transformer Block

```
Code here
```

Structure of Full Transformer



- A Token Embedding module
- Several Transformer Block modules
- A RMSNorm module
- Final Linear Transformation module

Assembly of Complete Transformer

```
Code here
```

```
Code here
```

Assembly of Complete Transformer

```
Code here
```