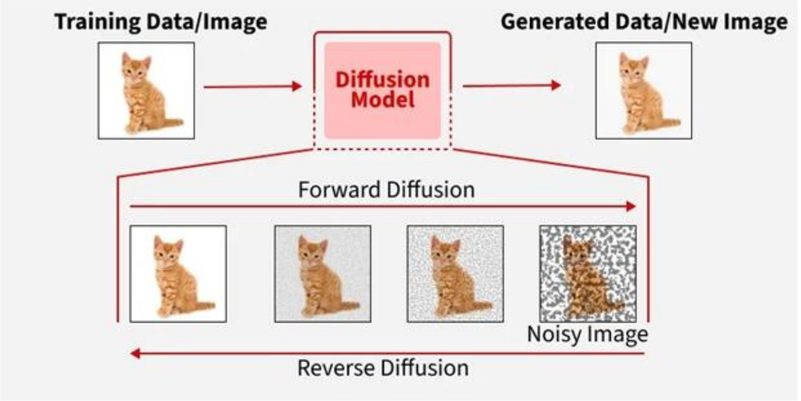
RICE UNIVERSITY

Week-6:
Transformer Operation

Transformer and Its Applications

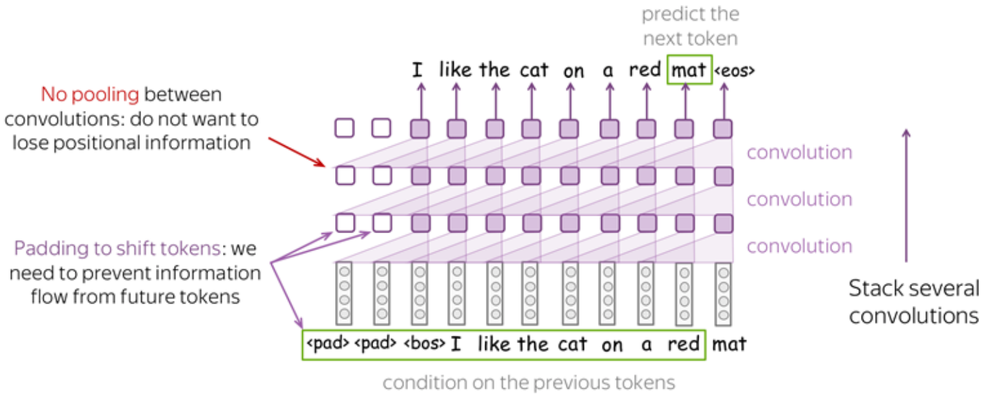
Gemini



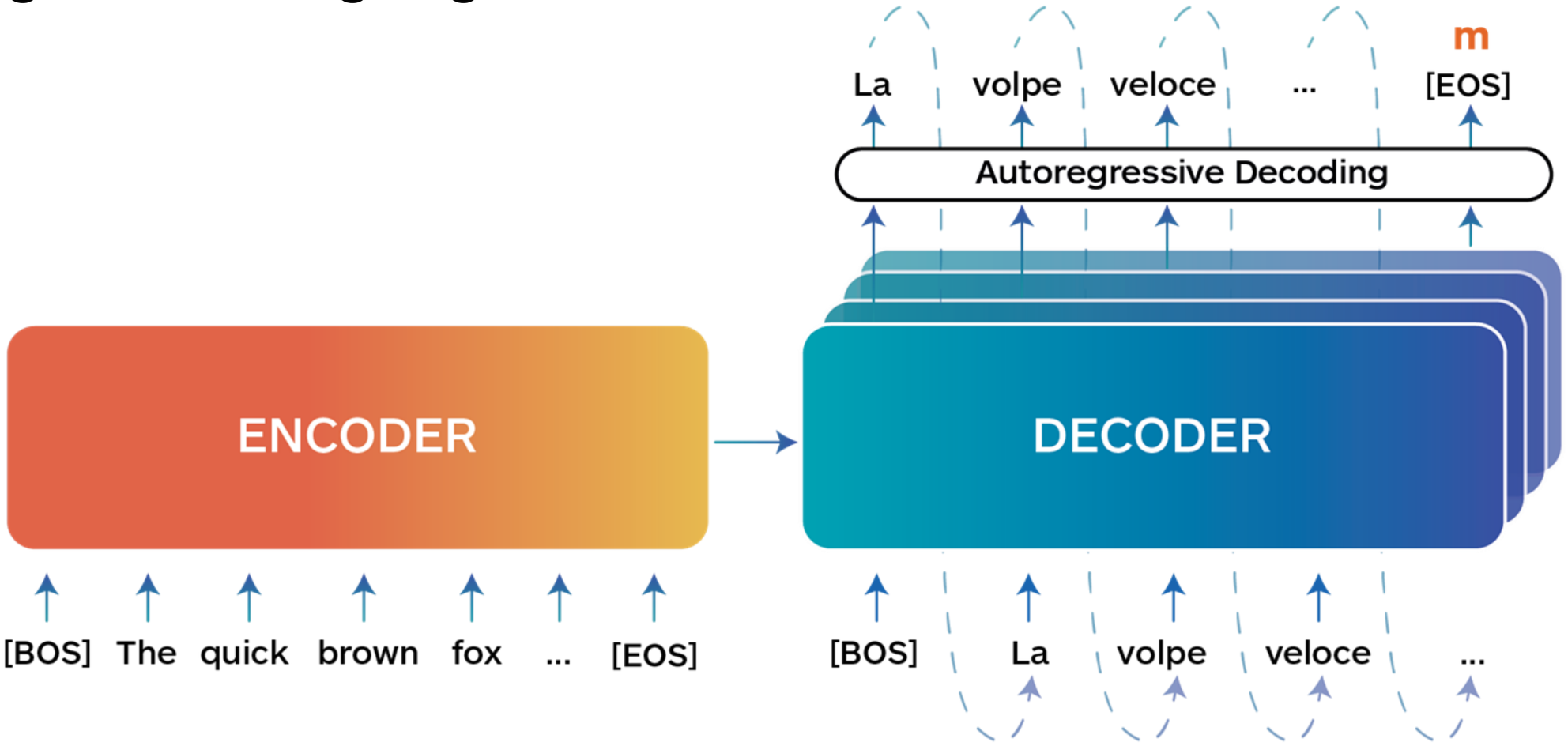
ChatGPT

Qwen3

deepseek

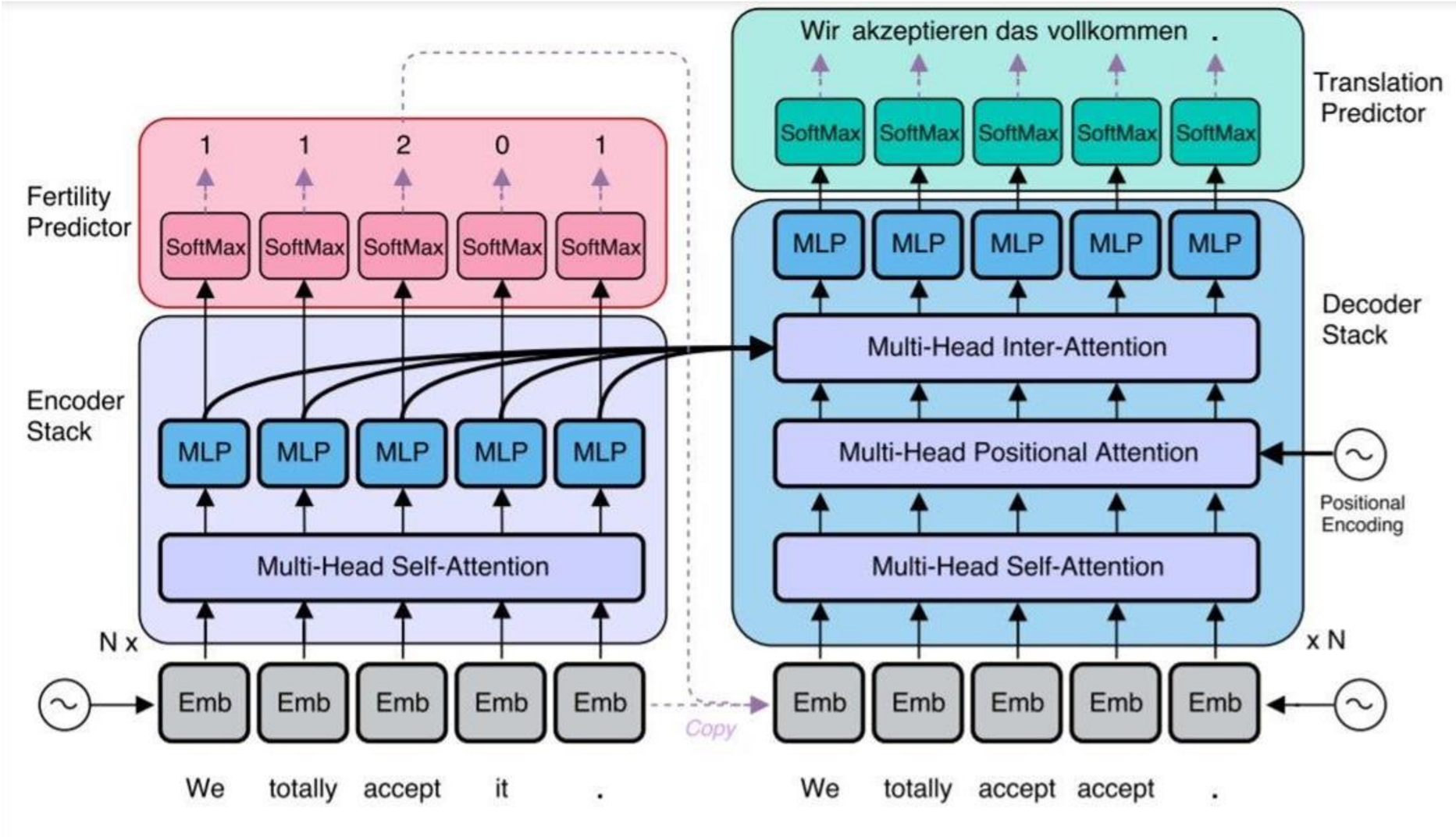


Autoregressive Language Model

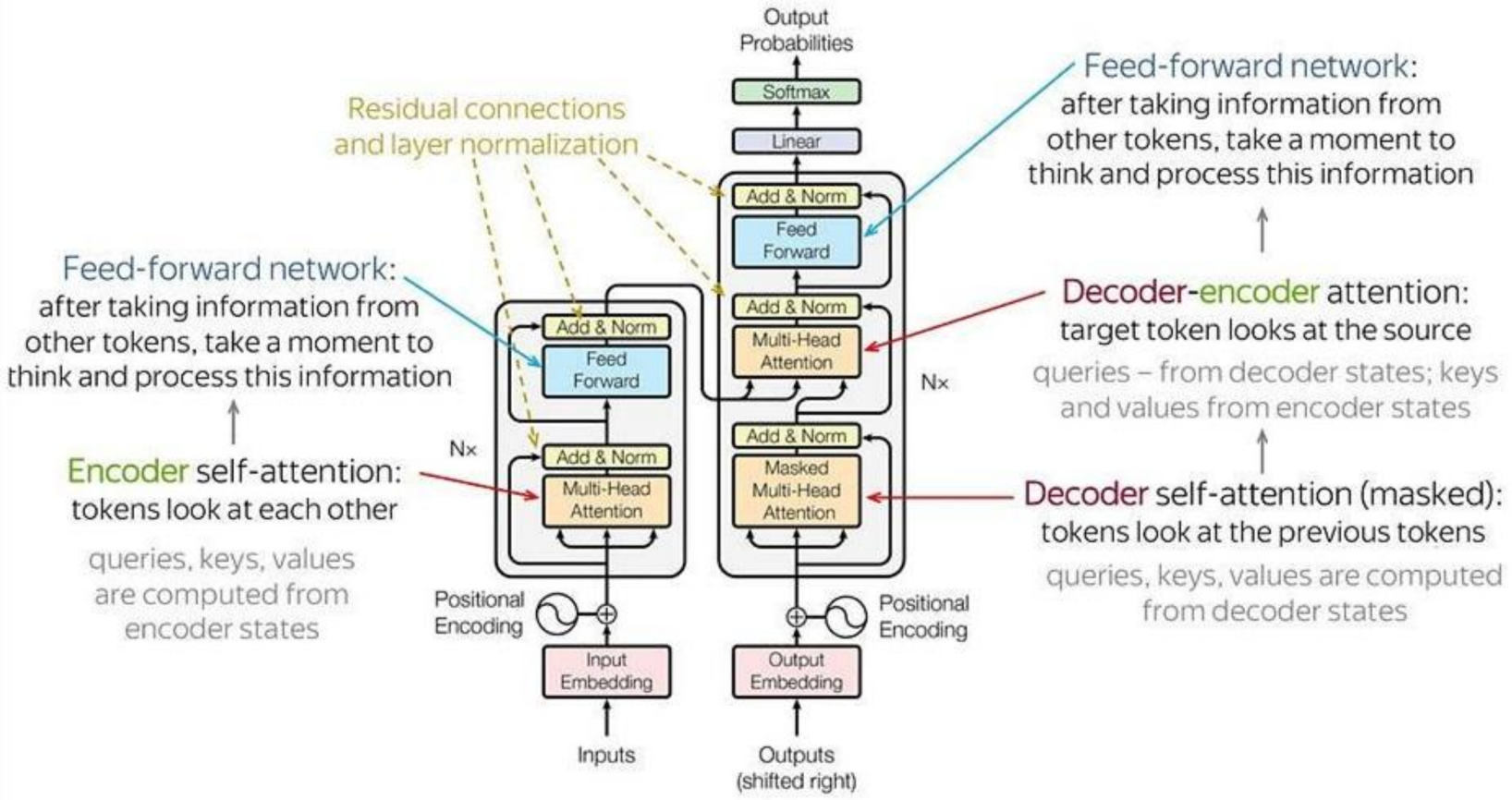


Standard generation procedure in Machine Translation:
Autoregressive Decoding. Each “word” is generated sequentially after the previous one.

Autoregressive Language Model



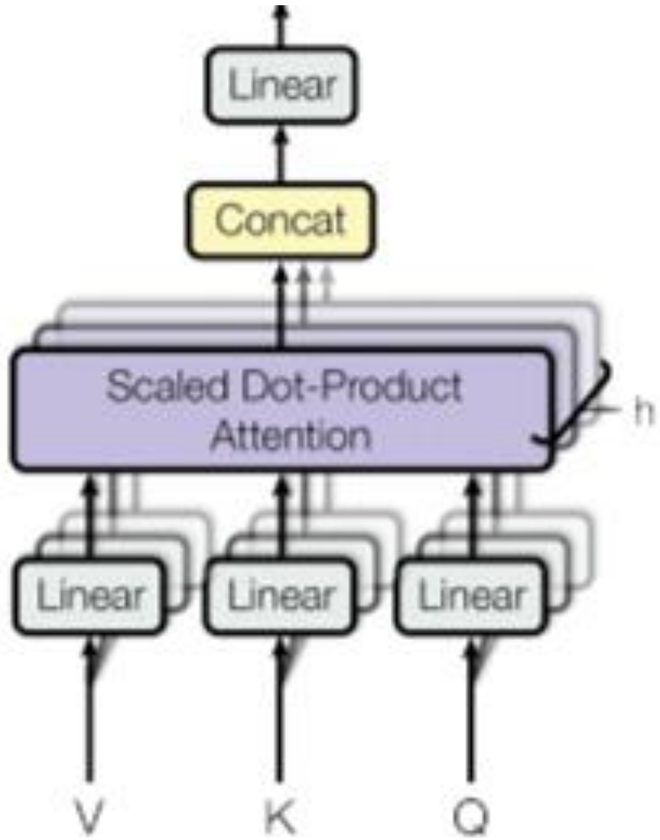
Transformer: The Building Block

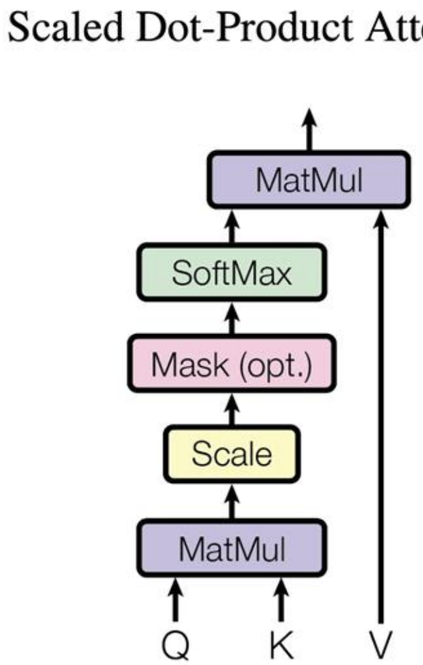


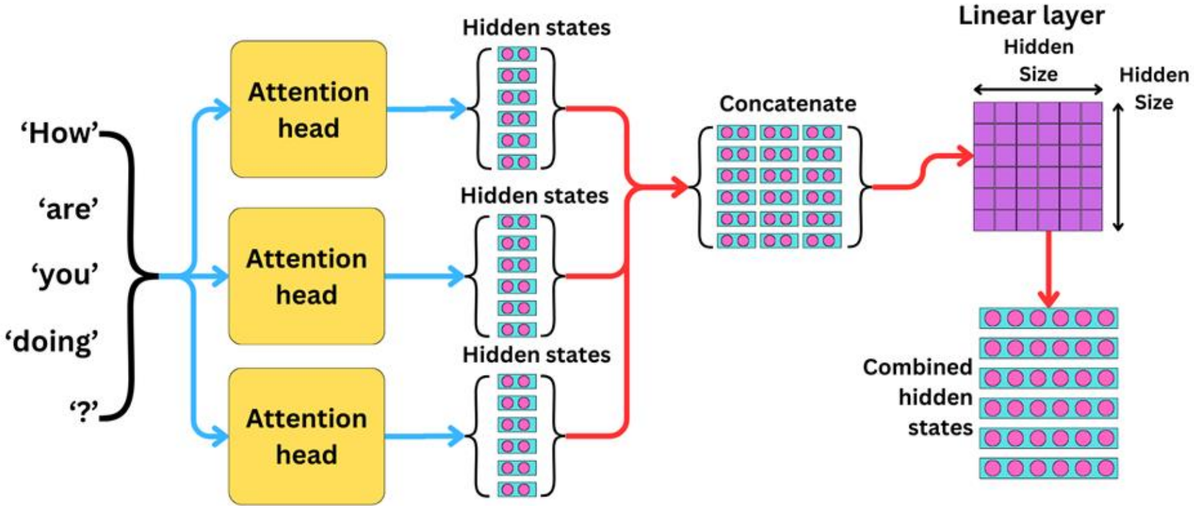
Input Sequence

A sequence of words or tokens that the model will process.

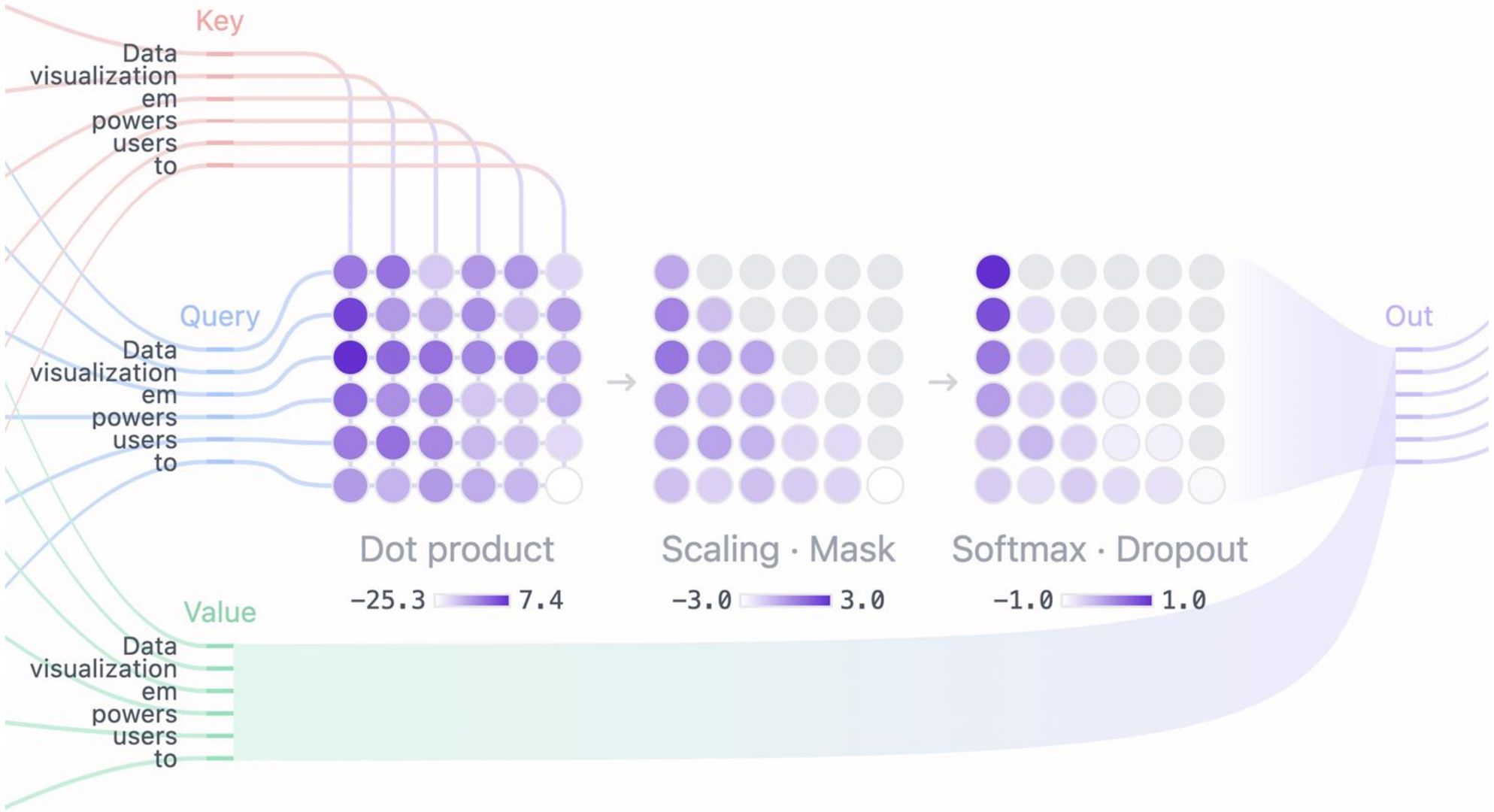
Multi-head Attention Block



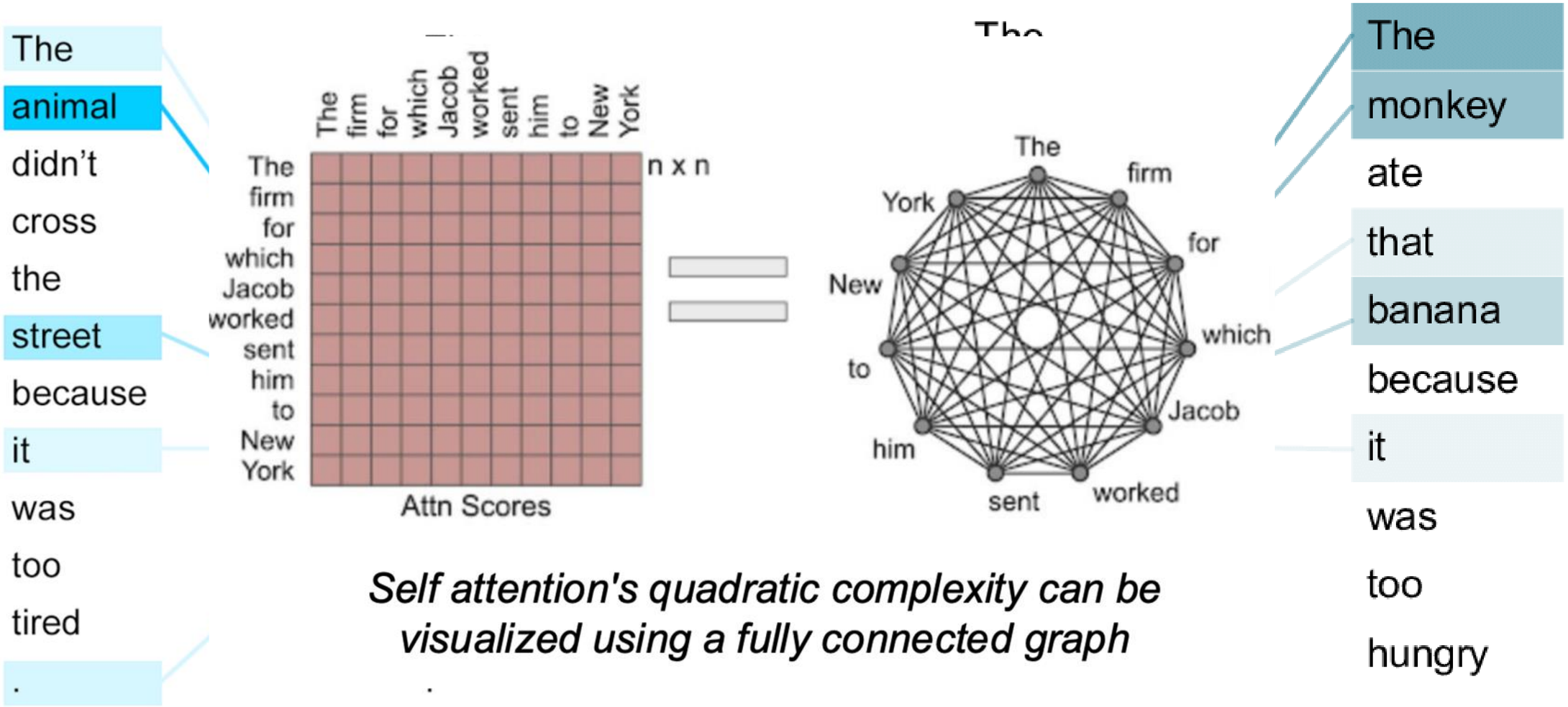




Multi-head Attention Block

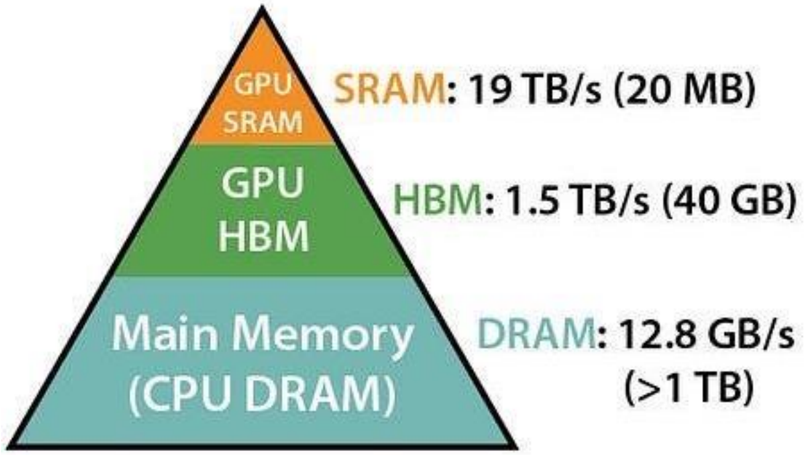


FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

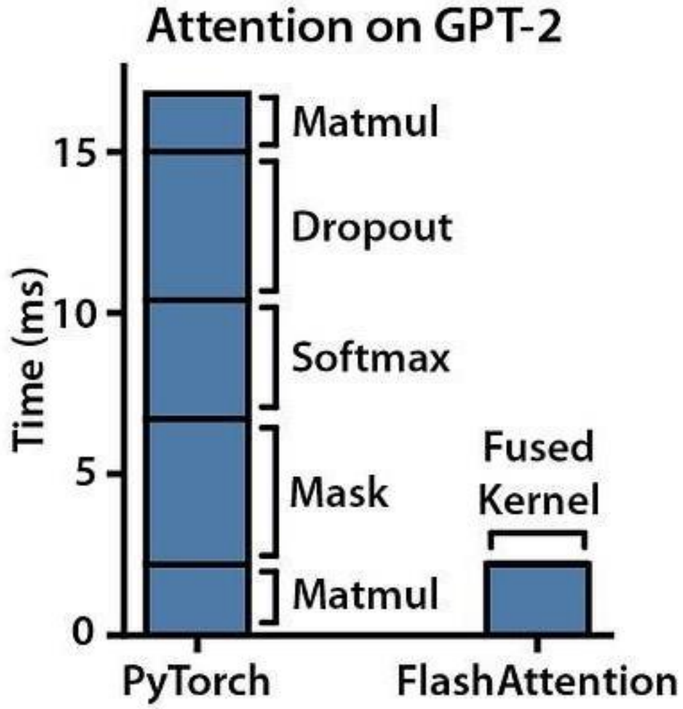


Self attention's quadratic complexity can be visualized using a fully connected graph

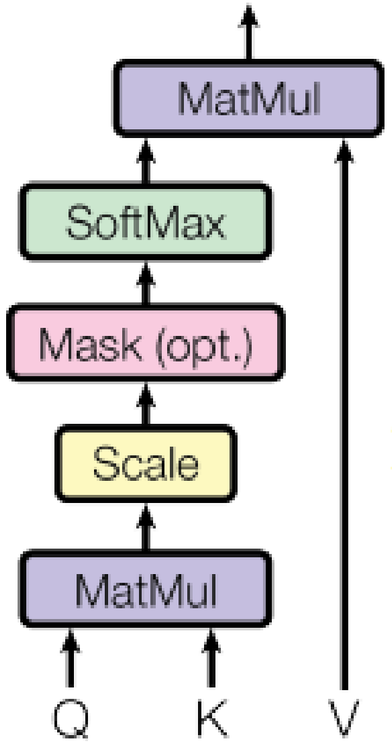
FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness



Memory Hierarchy with Bandwidth & Memory Size



Standard Attention Computation



Given input sequences $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ where N is the sequence length and d is the head dimension, we want to compute the attention output $\mathbf{O} \in \mathbb{R}^{N \times d}$:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d},$$

where softmax is applied row-wise.

Standard Attention Computation

- Standard Softmax

$$m(x) := \max_i \; x_i, \quad f(x) := \begin{bmatrix} e^{x_1 - m(x)} & \dots & e^{x_B - m(x)} \end{bmatrix}, \quad \ell(x) := \sum_i f(x)_i, \quad \text{softmax}(x) := \frac{f(x)}{\ell(x)}$$

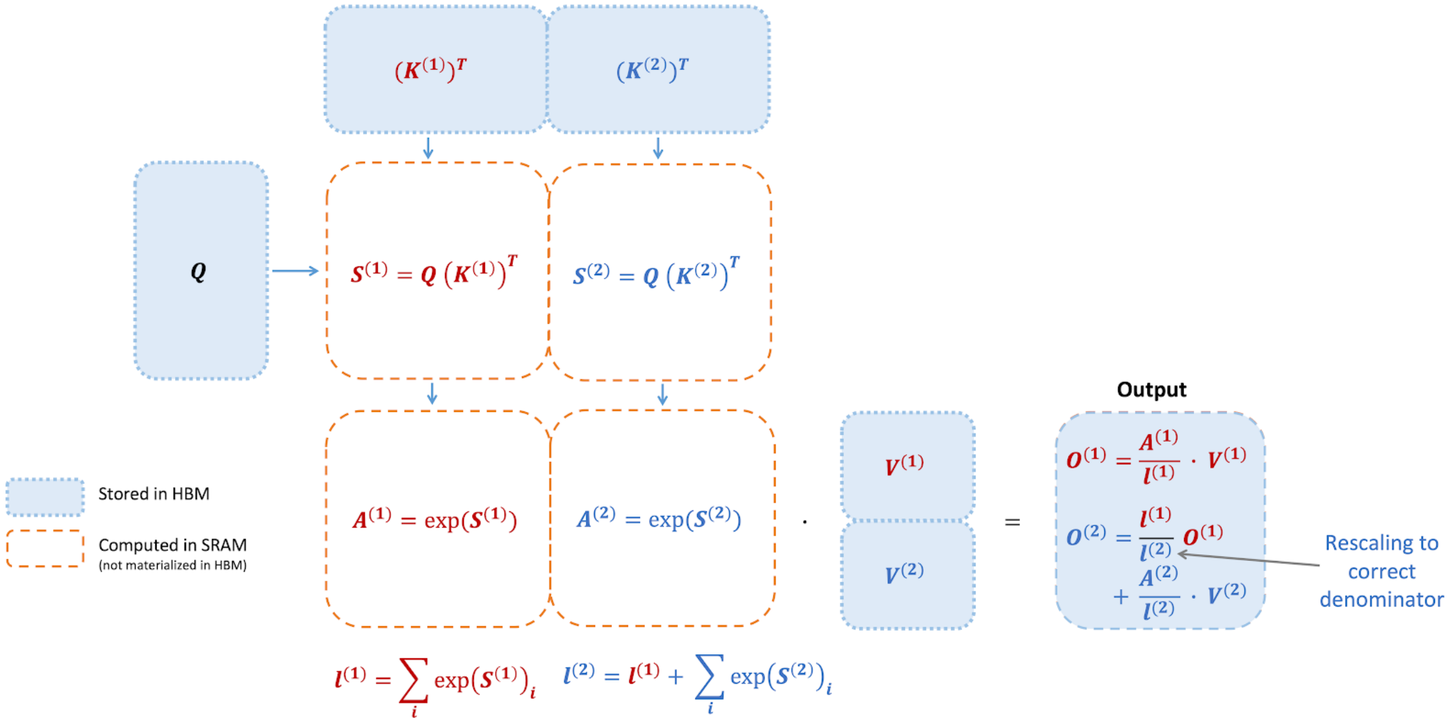
- Block/tiled Softmax

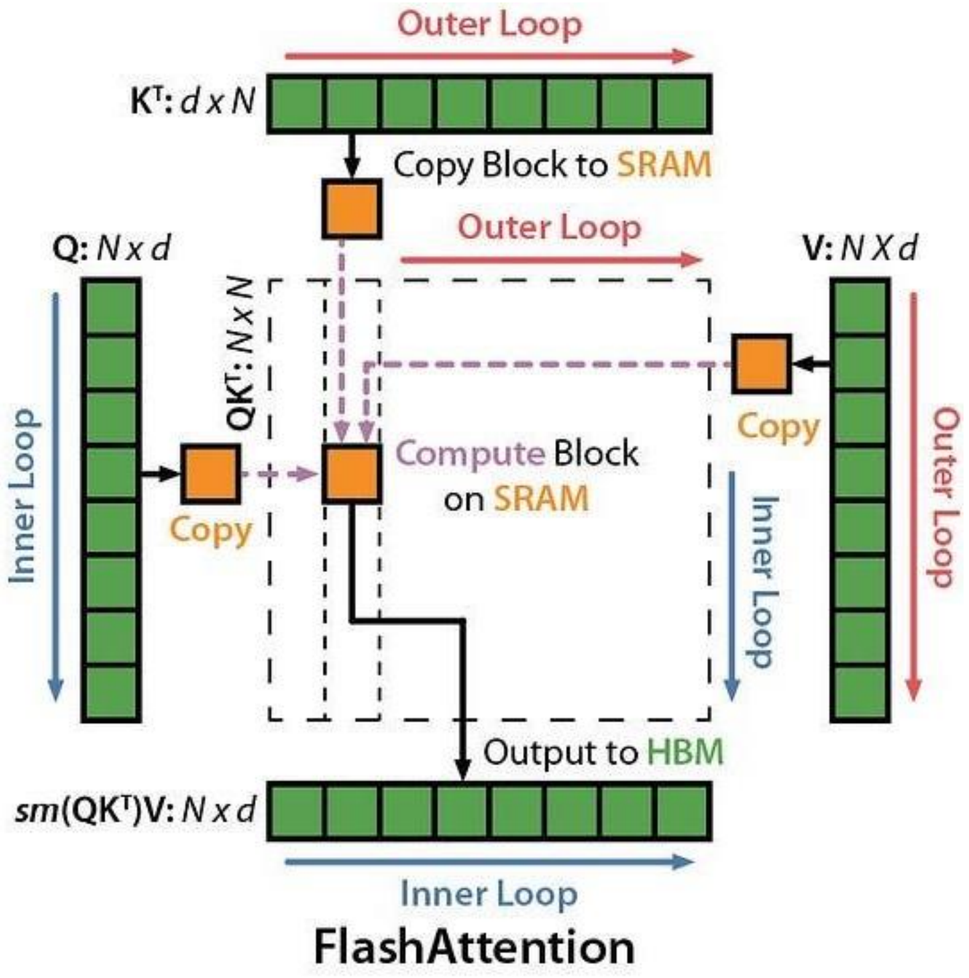
For vectors $x^{(1)}, x^{(2)} \in \mathbb{R}^B$, we can decompose the softmax of the concatenated $x = \begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix} \in \mathbb{R}^{2B}$ as:

$$m(x) = m(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = \max(m(x^{(1)}), m(x^{(2)})), \quad f(x) = \begin{bmatrix} e^{m(x^{(1)}) - m(x)} f(x^{(1)}) & e^{m(x^{(2)}) - m(x)} f(x^{(2)}) \end{bmatrix},$$

$$\ell(x) = \ell(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = e^{m(x^{(1)}) - m(x)} \ell(x^{(1)}) + e^{m(x^{(2)}) - m(x)} \ell(x^{(2)}), \quad \text{softmax}(x) = \frac{f(x)}{\ell(x)}.$$

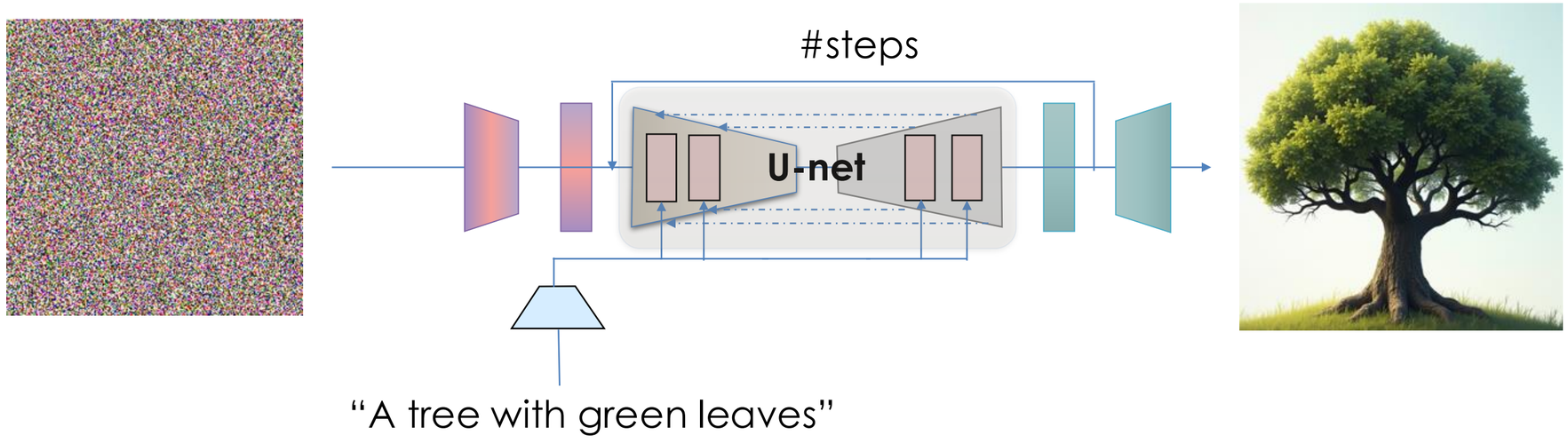
FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness





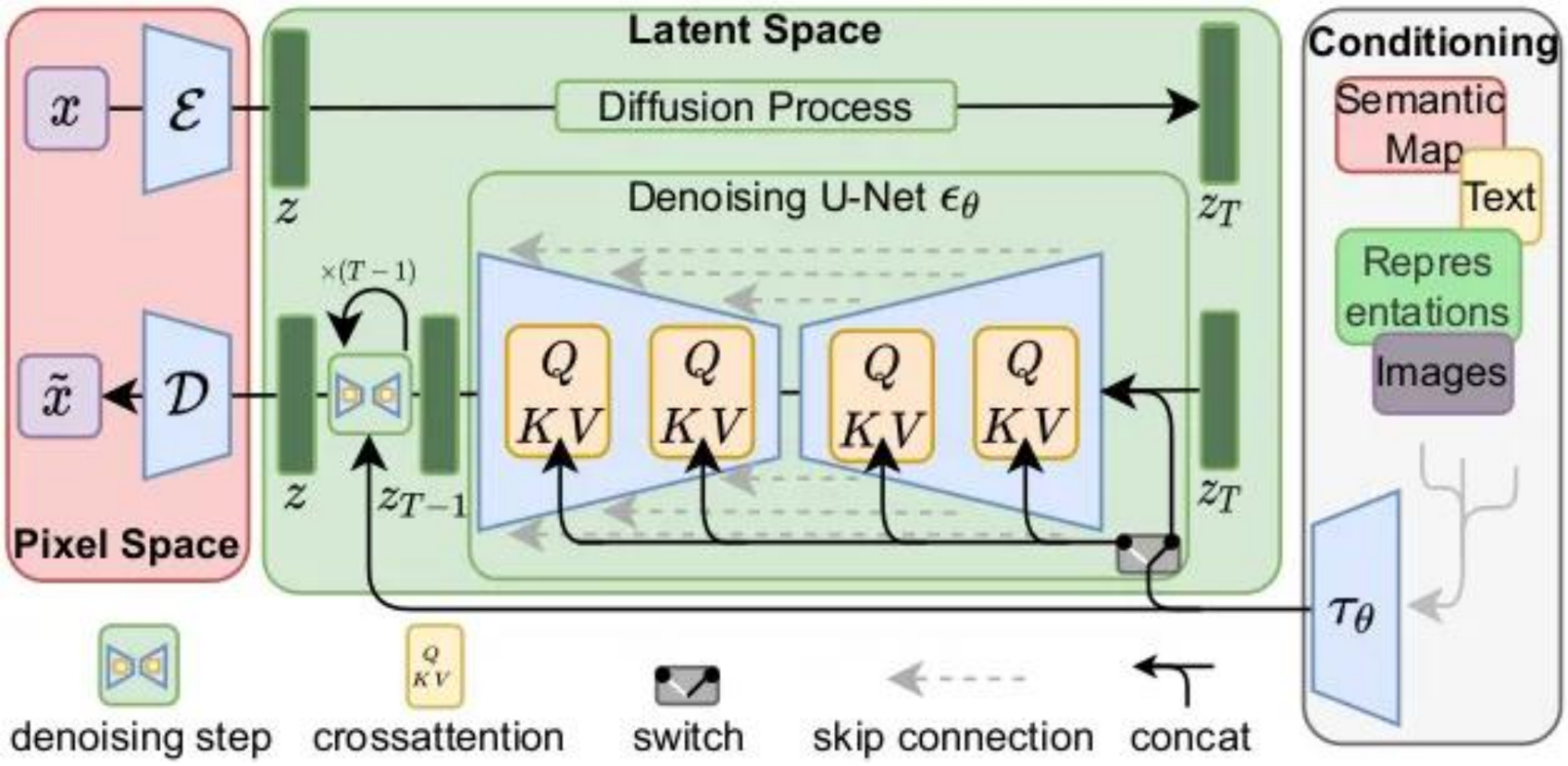
Diffusion Model for Image Generation

- Architecture of Stable Diffusion Model.

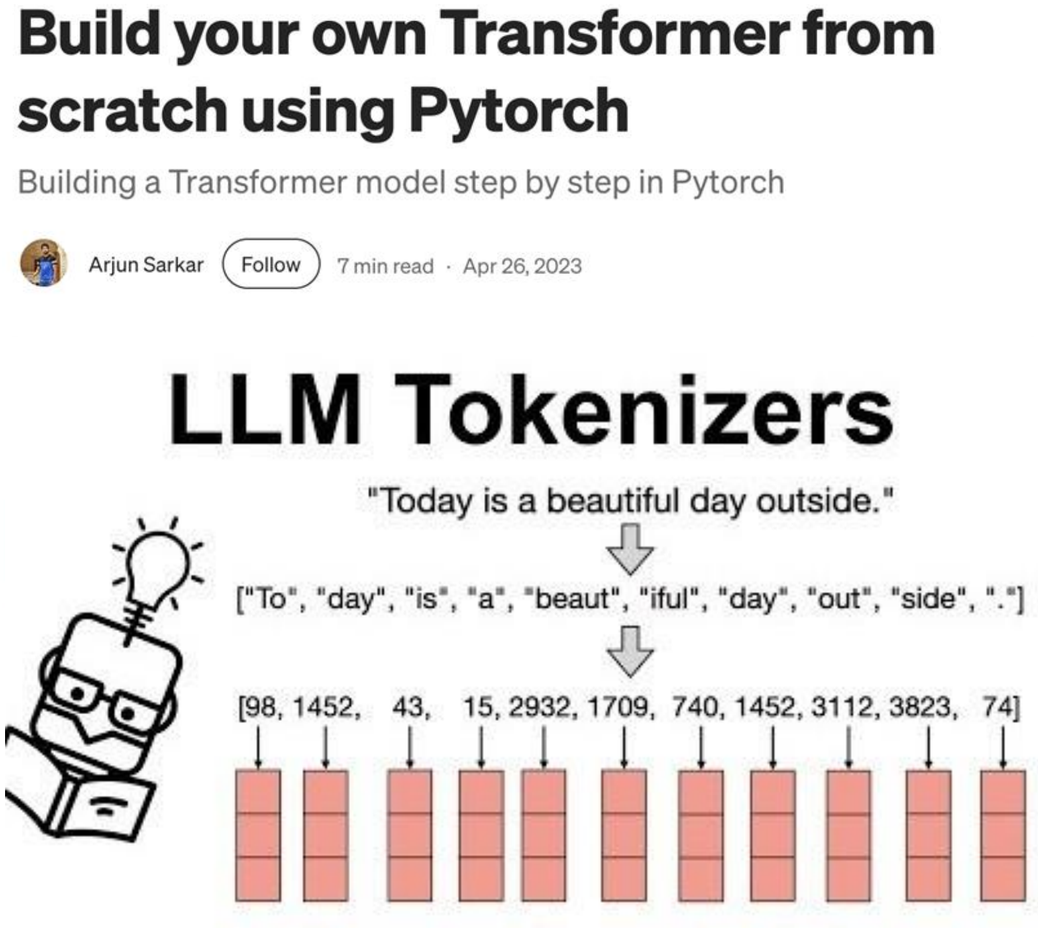


Note that Conv are omitted in Unit for simplicity.

Diffusion Transformer (DiT)



Transformer Implementation



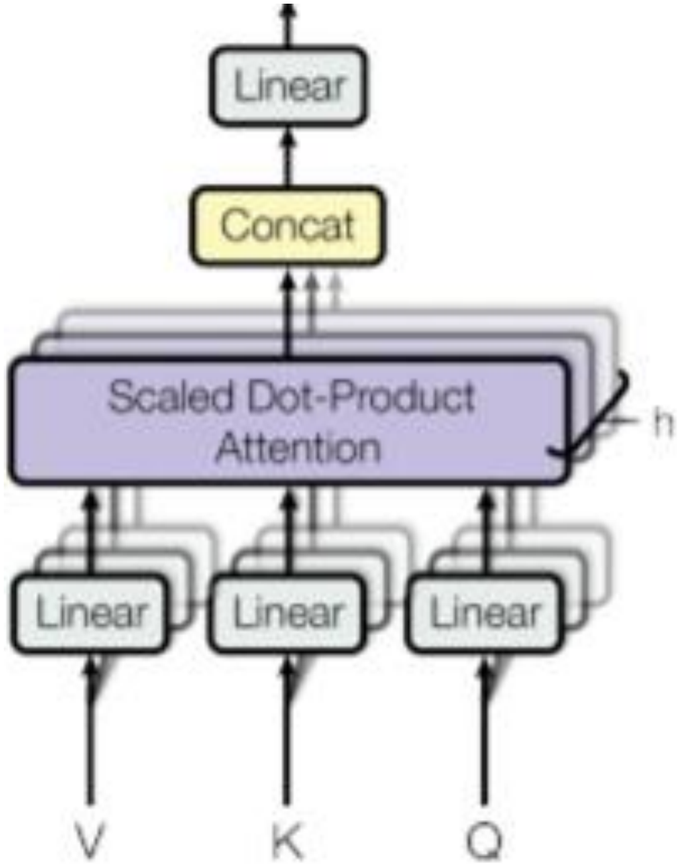
1. Data Preparation:

- **Tokenization:** Convert raw text into numerical tokens (e.g., using Byte-Pair Encoding or WordPiece).
- **Embedding:** Map tokens to dense vector representations, capturing semantic meaning.
- **Positional Encoding:** Add positional information to the embeddings, as Transformers do not inherently process sequences in order. This can be done using sine and cosine functions.

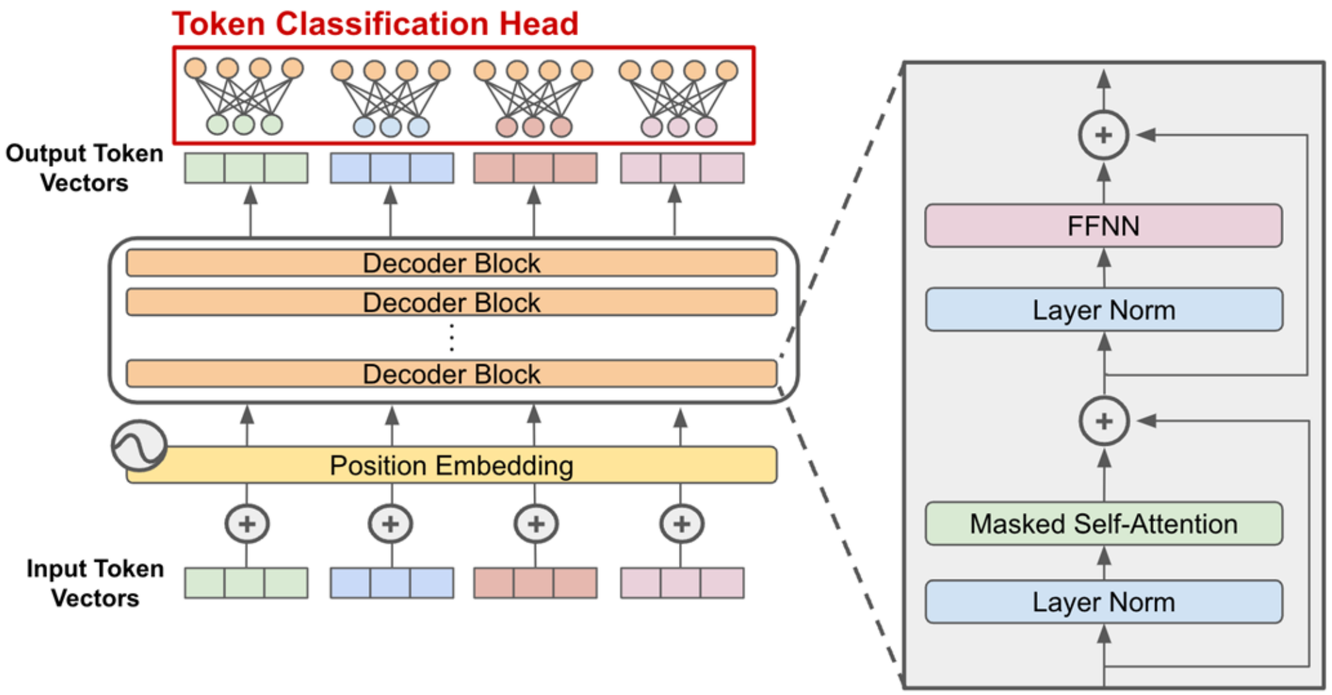
•**Add & Norm:** Uses **residual connections** and **layer normalization** to stabilize training.
•**Layer Stacking:** Repeats this entire process multiple times to deepen feature extraction.

Transformer Implementation

- **Multi-Head Attention:** Maps relationships between tokens using Q, K, and V matrices across parallel "heads."
- **Feed-Forward Network:** Applies independent transformations to each position via a two-layer network.
- **Add & Norm**: Uses residual connections and layer normalization to stabilize training.
- **Layer Stacking:** Repeats this entire process multiple times to deepen feature extraction.



Transformer Implementation




```python
import torch
import torch.nn as nn

class MultiHeadSelfAttention(nn.Module):
    # ... implementation of Q, K, V, dot product attention, heads ...

class PositionwiseFeedForward(nn.Module):
    # ... implementation of two linear layers with activation ...

class EncoderLayer(nn.Module):
    # ... combines MultiHeadSelfAttention and PositionwiseFeedForward with

class Encoder(nn.Module):
    # ... stacks multiple EncoderLayers ...

class DecoderLayer(nn.Module):
    # ... combines Masked MultiHeadSelfAttention, MultiHeadCrossAttention,

class Decoder(nn.Module):
    # ... stacks multiple DecoderLayers ...

class Transformer(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size, d_model, n_heads, n_
        super().__init__()
        # ... embedding layers, positional encoding ...
        self.encoder = Encoder(...)
        self.decoder = Decoder(...)
        self.final_linear = nn.Linear(d_model, tgt_vocab_size)

    def forward(self, src, tgt, src_mask, tgt_mask):
        # ... forward pass through encoder, decoder, and final linear layer
        return output
```

RICE UNIVERSITY

yuke.wang@rice.edu