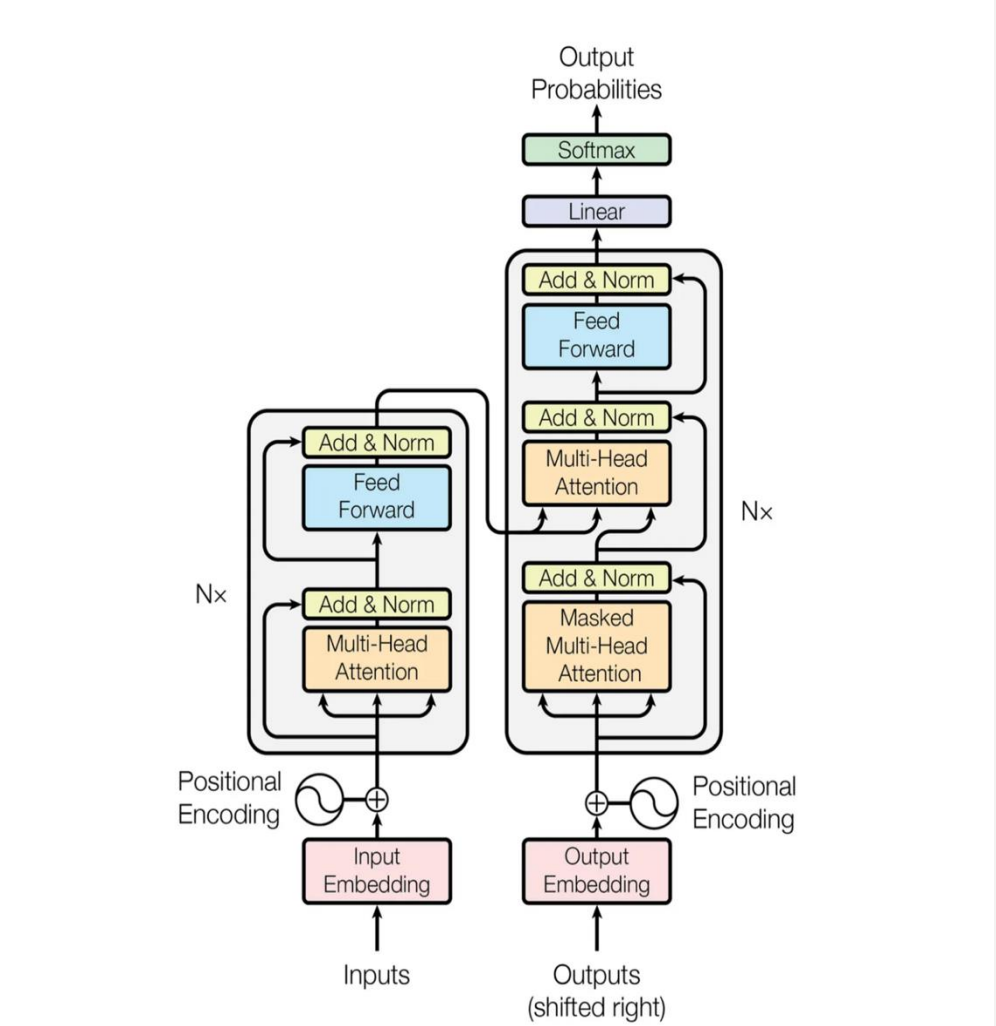
TRANSFORMERS

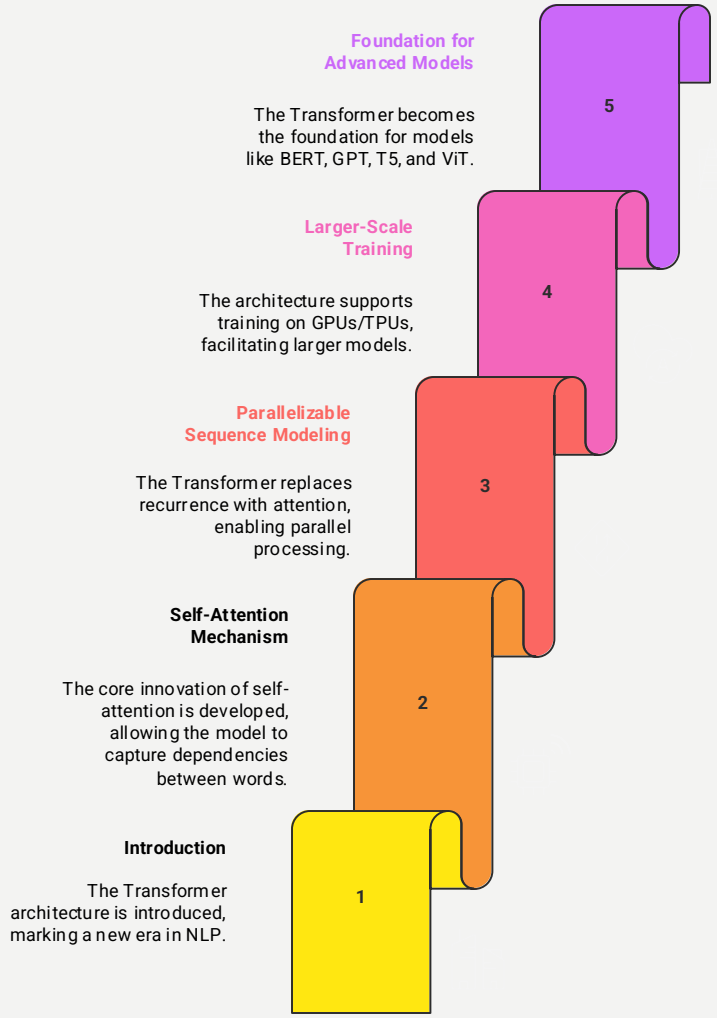
TRANSFORMER ARCHITECTURE



TRANSFORMERS

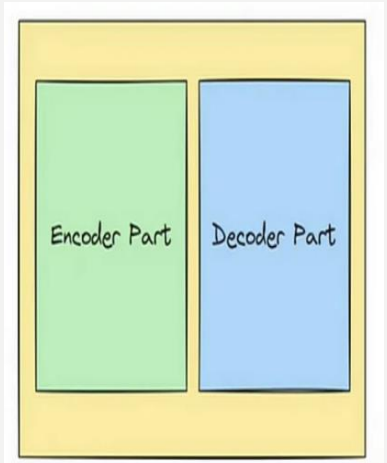
- **1. Introduction**
- The **Transformer** is a deep learning architecture introduced in 2017 that revolutionized Natural Language Processing (NLP).
  Unlike RNNs or CNNs, it does **not rely on recurrence or convolution**; instead, it leverages a mechanism called **self-attention** to capture dependencies between words in a sequence, regardless of their distance.
- **Key Innovations:**
- Replaces recurrence with **attention** for parallelizable sequence modeling.
- Enables much larger-scale training via GPUs/TPUs.
- Forms the foundation for models like **BERT**, **GPT**, **T5**, and **ViT**.

Transformer's Journey to Revolutionize NLP



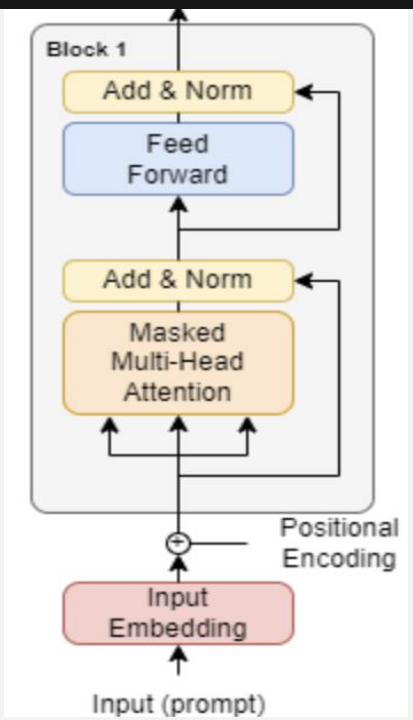
TRANSFORMER ARCHITECTURE

- **2. Model Overview**
- The Transformer model consists of:
- **Encoder:** Processes input sequence and produces contextual embeddings.
- **Decoder:** Generates output sequence (e.g., translated sentence).
- Each consists of **N identical layers** (typically 6 in the original paper).
- **Architecture Summar**

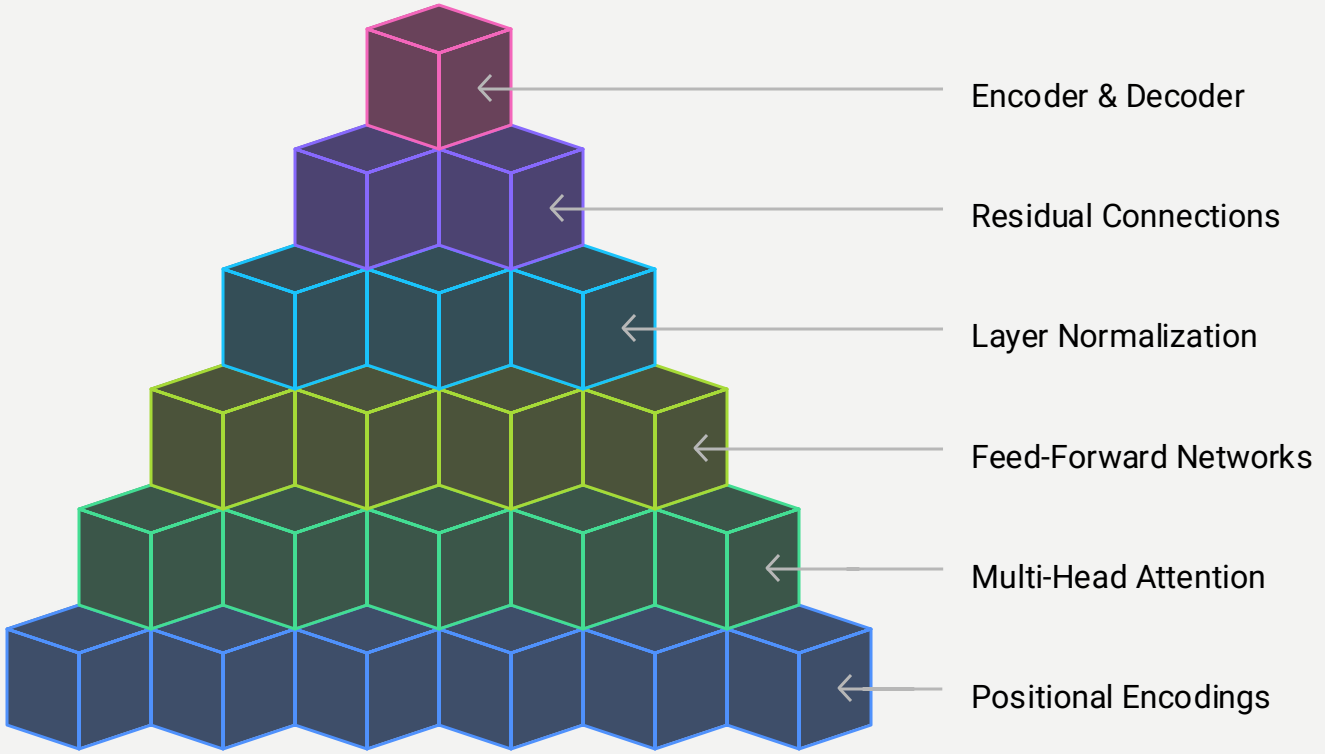


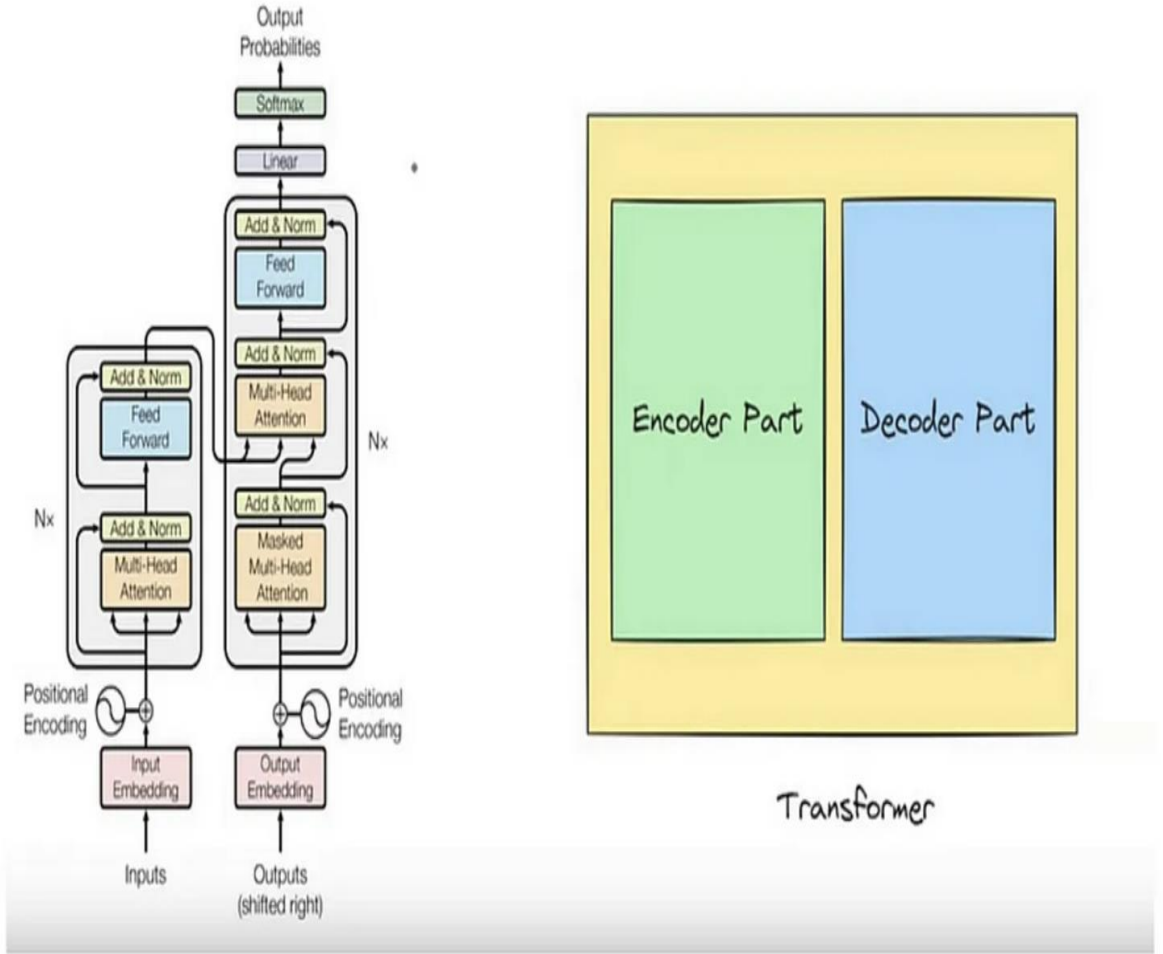

```
Input → Encoder Stack → Encoder Output → Decoder Stack → Linear + Softmax → Output Tokens
```

- **Core Components:**
- Multi-Head Self-Attention
- Position-wise Feed-Forward Networks
- Positional Encodings
- Layer Normalization
- Residual Connections

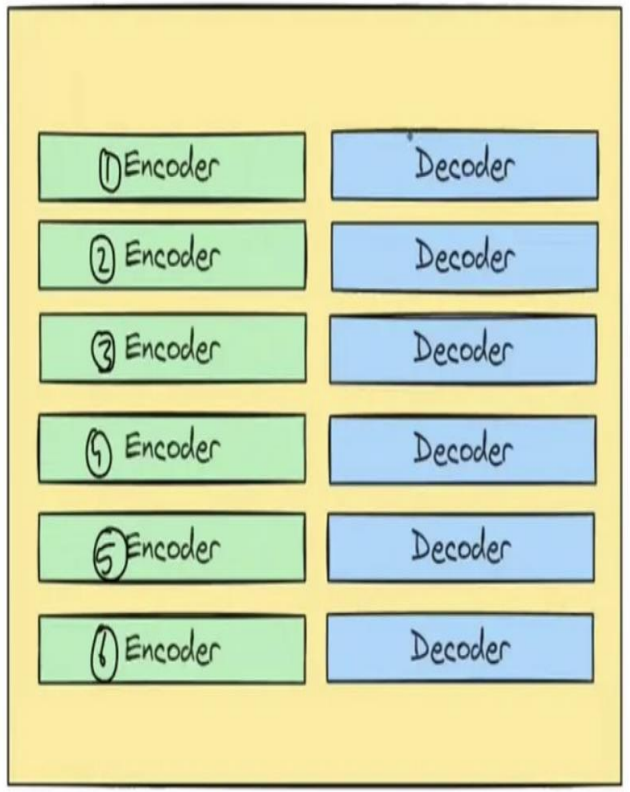


Transformer Model Hierarchy

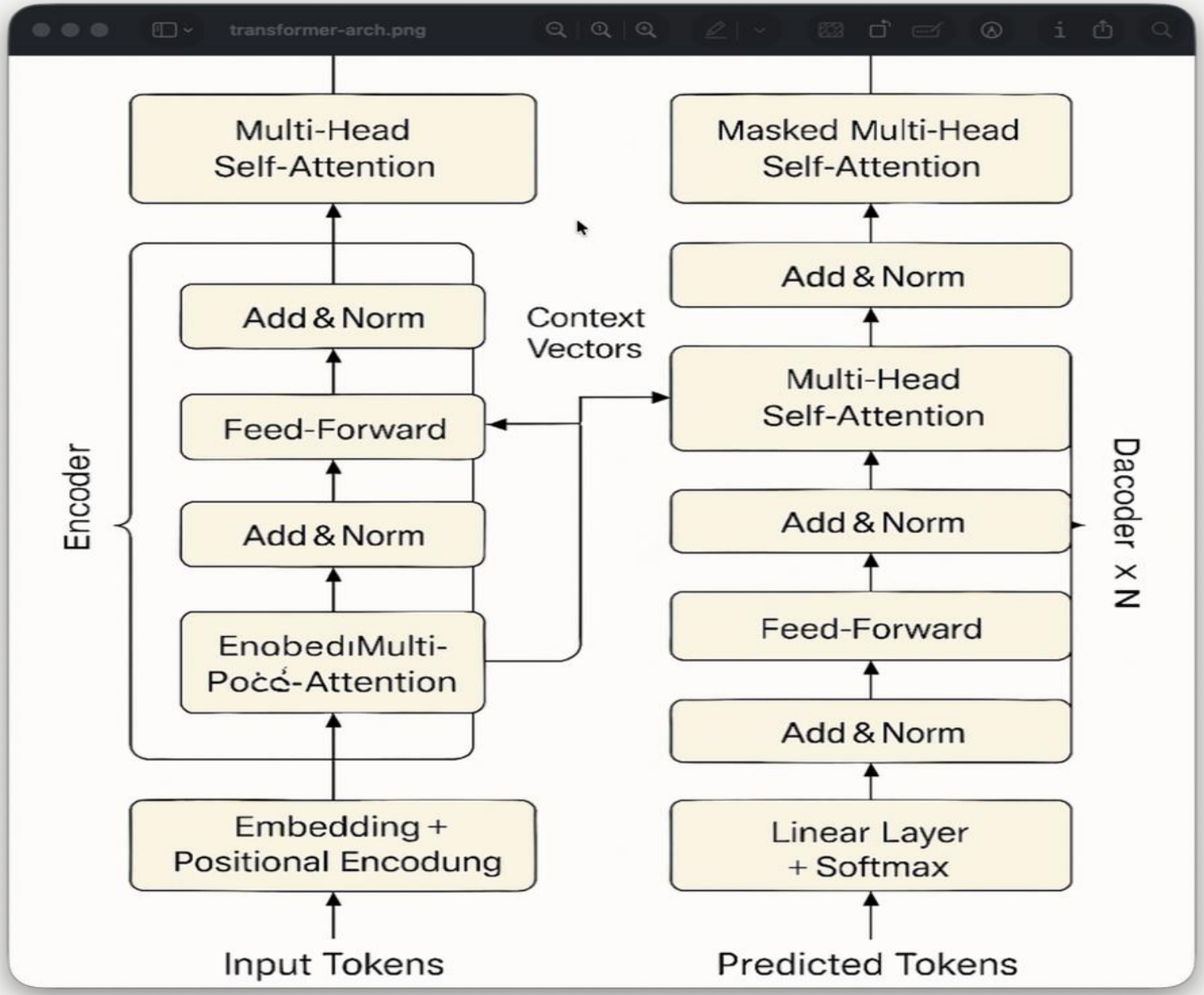


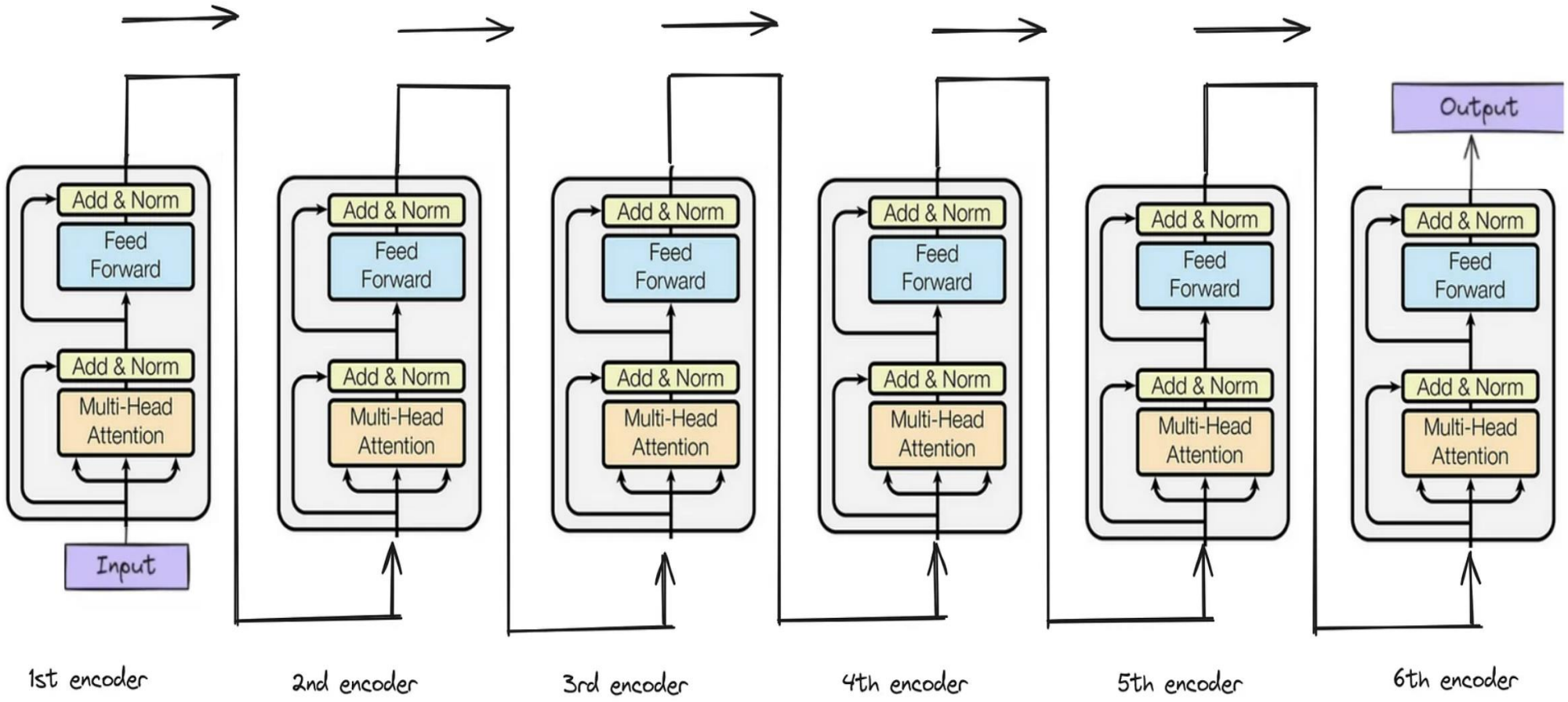


Transformer

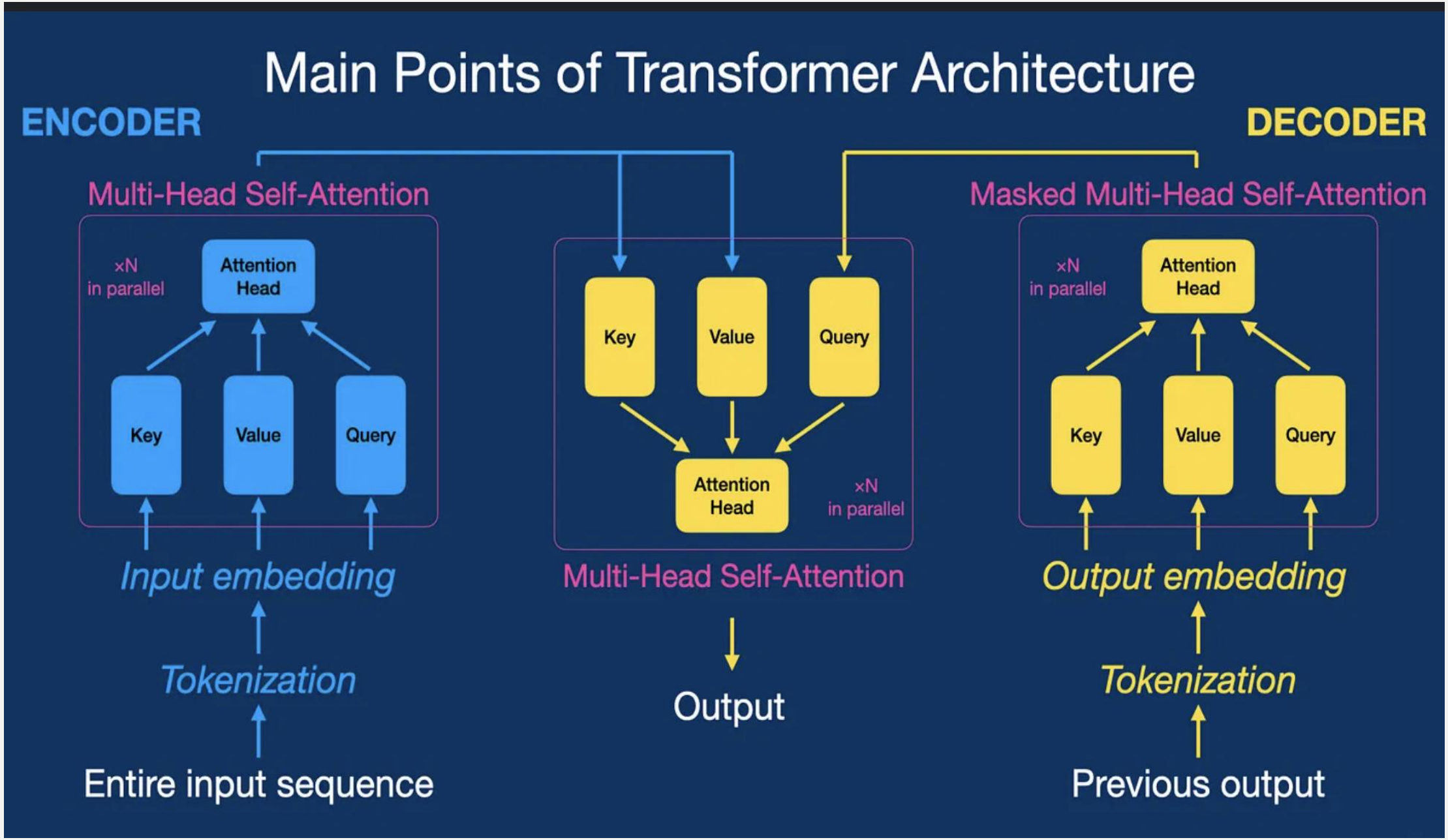


Transformer





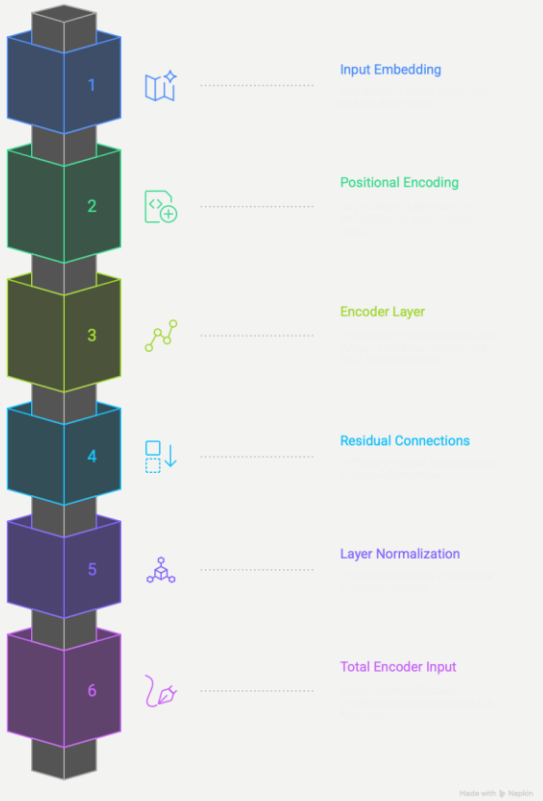
Flow of data : output of one encoder is input for next encoder block

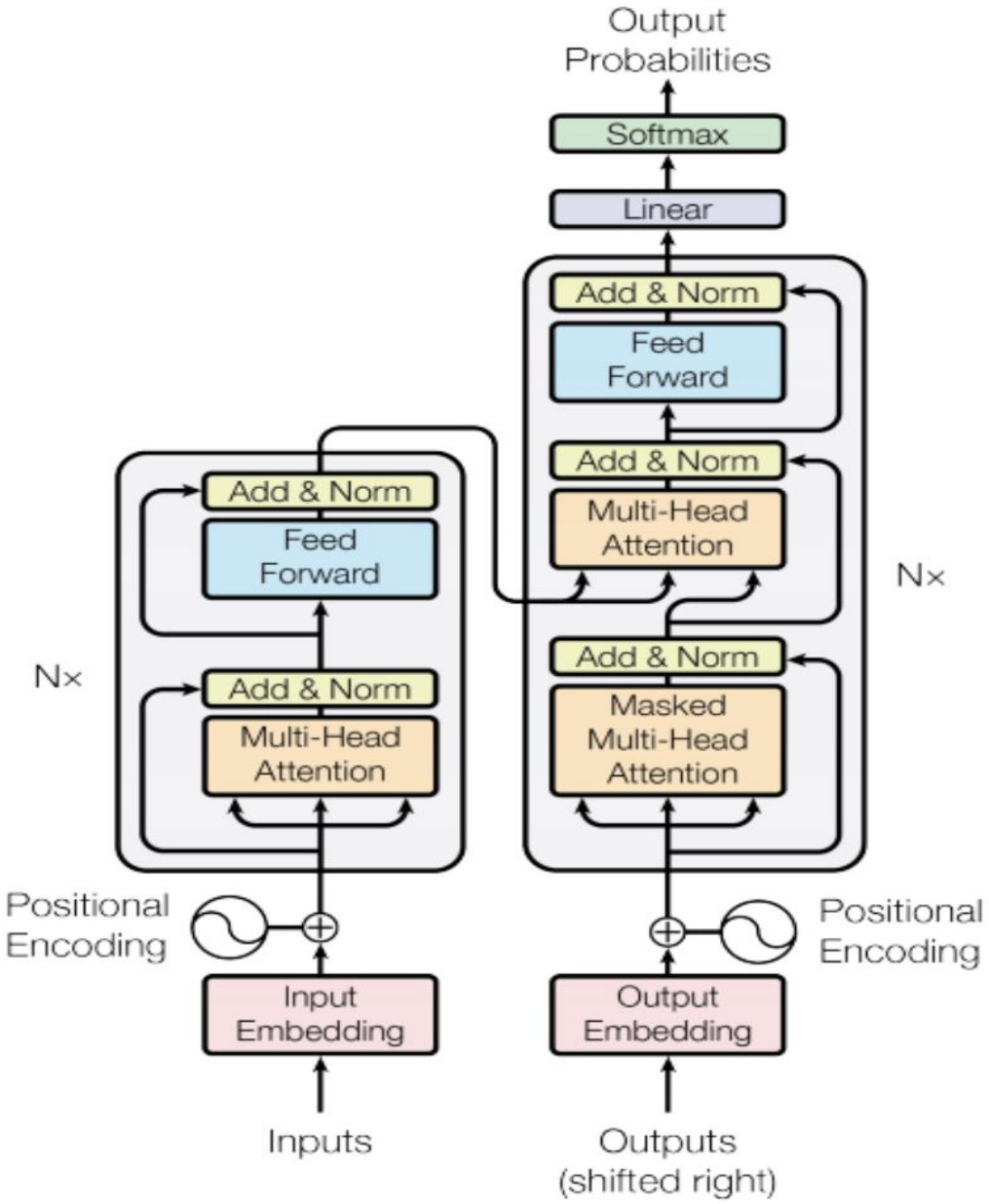


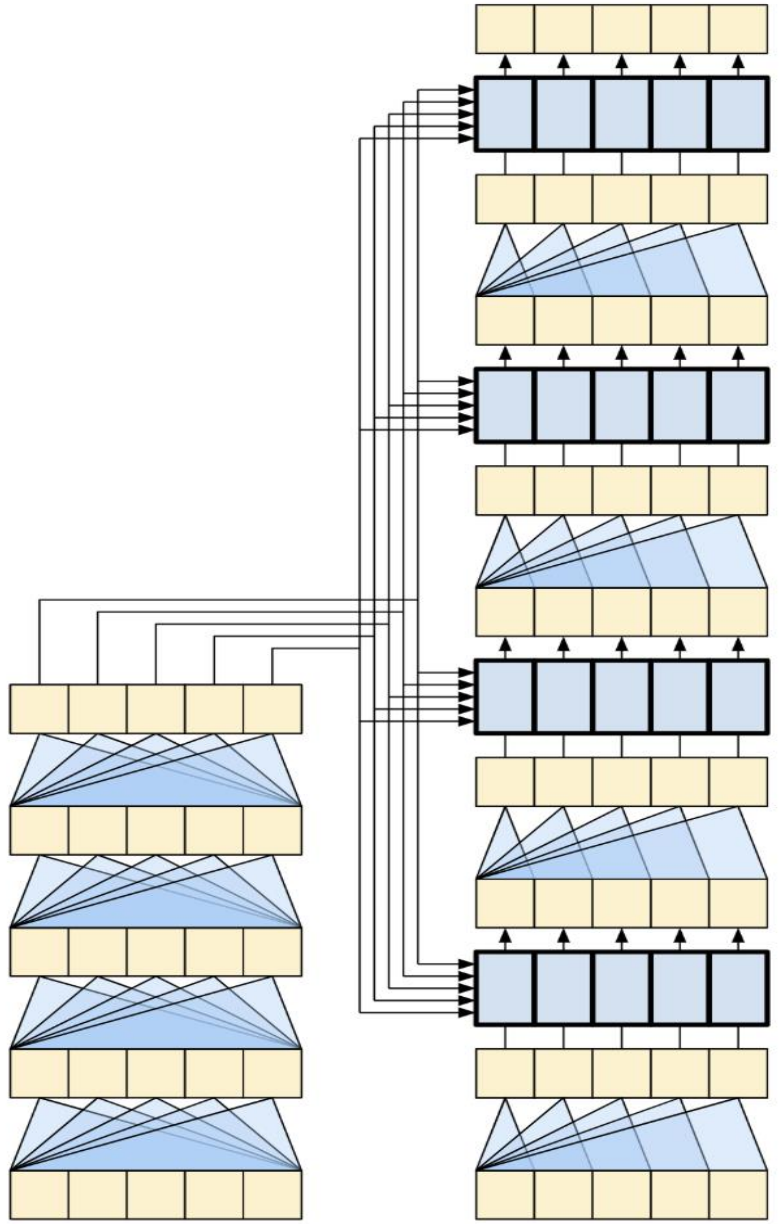
- **3. Encoder Architecture**
- Each **encoder layer** contains two main sub-layers:
- **Multi-Head Self-Attention**
- **Feed-Forward Network**
- Both use **residual connections** followed by **layer normalization**.
- **3.1 Input Embedding**
- Each token is mapped to a dense vector:
- $E = X \times W_e$
- where $X$is the one-hot representation and $W_e \in \mathbb{R}^{V \times d_{model}}$is the embedding matrix.
- **3.2 Positional Encoding**
- Since there's no recurrence, positional encodings are added to embeddings:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

- This gives the model a notion of token order.
- **Total Encoder Input:**
- $Z_0 = E + PE$

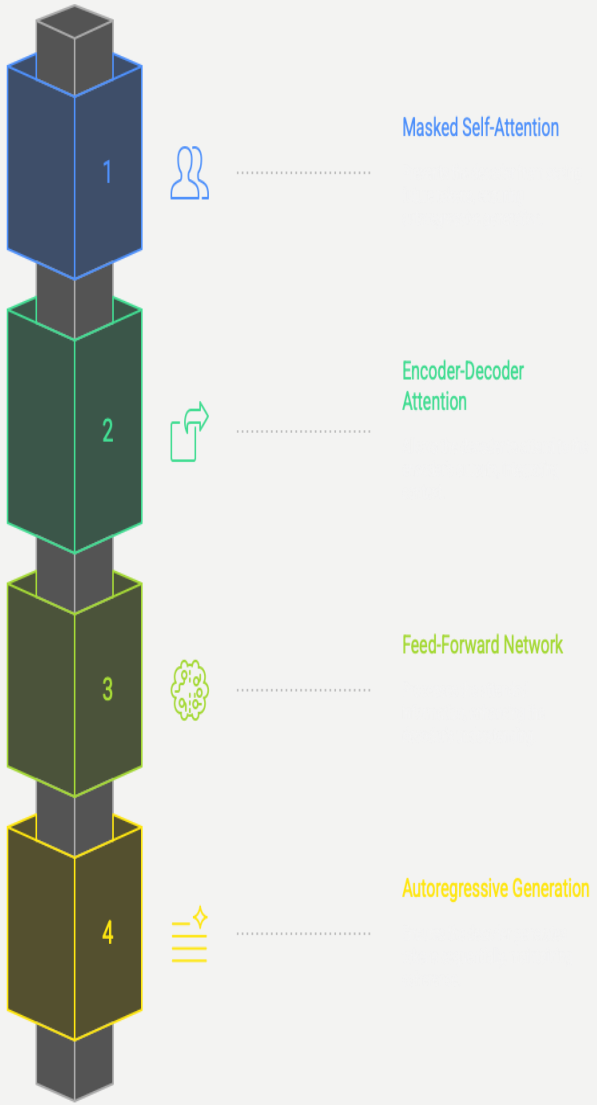




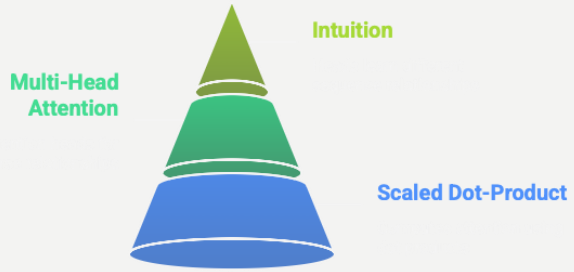


- **4. Decoder Architecture**
- The **decoder** mirrors the encoder but adds:
- **Masked Multi-Head Self-Attention** (prevents seeing future tokens)
- **Encoder–Decoder Attention** (attends to encoder outputs)
- **Feed-Forward Network**
- **Masking** ensures autoregressive generation:

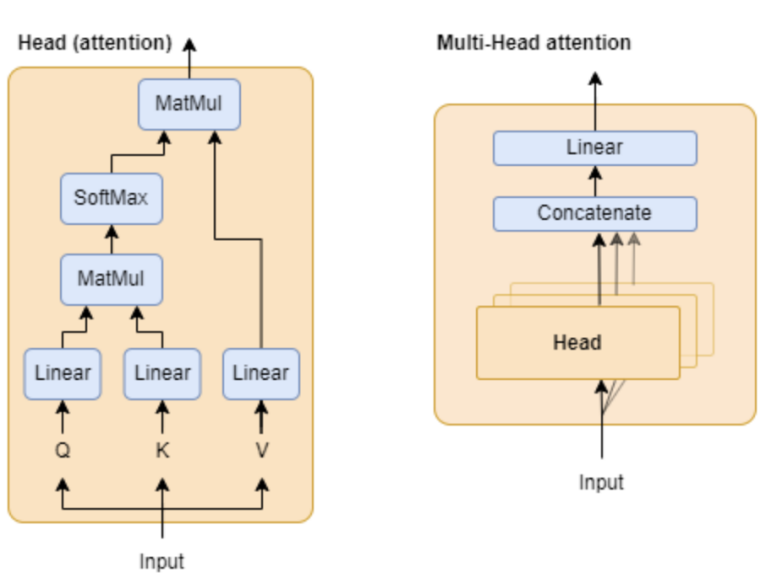
$$\text{Mask}(QK^T)_{ij} = \begin{cases} -\infty, & \text{if } j > i \\ QK^T_{ij}, & \text{otherwise} \end{cases}$$

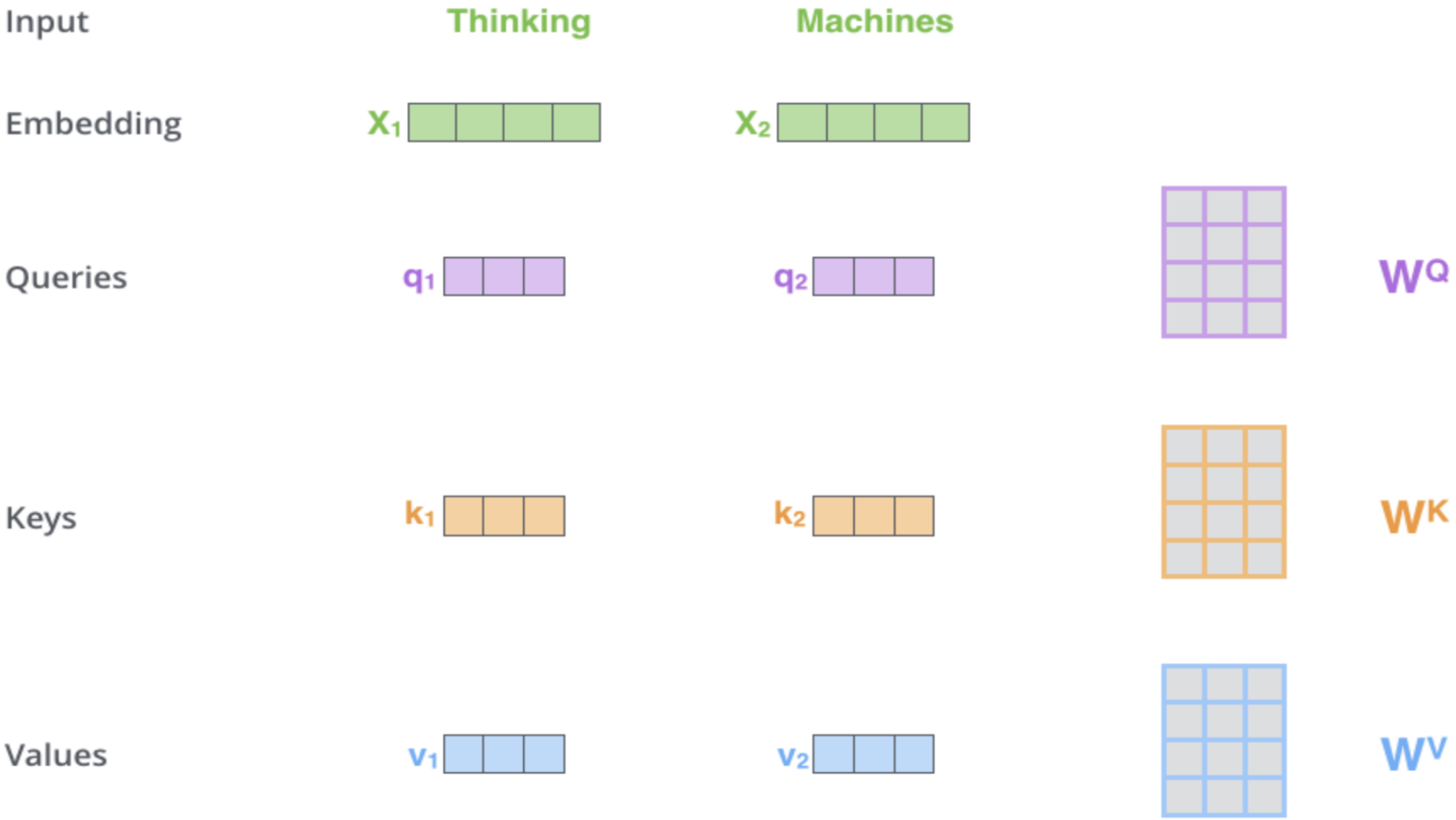


- **5. Attention Mechanism**
- **5.1 Scaled Dot-Product Attention**
- Given Query (Q), Key (K), and Value (V) matrices:
- $\text{Attention}(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$

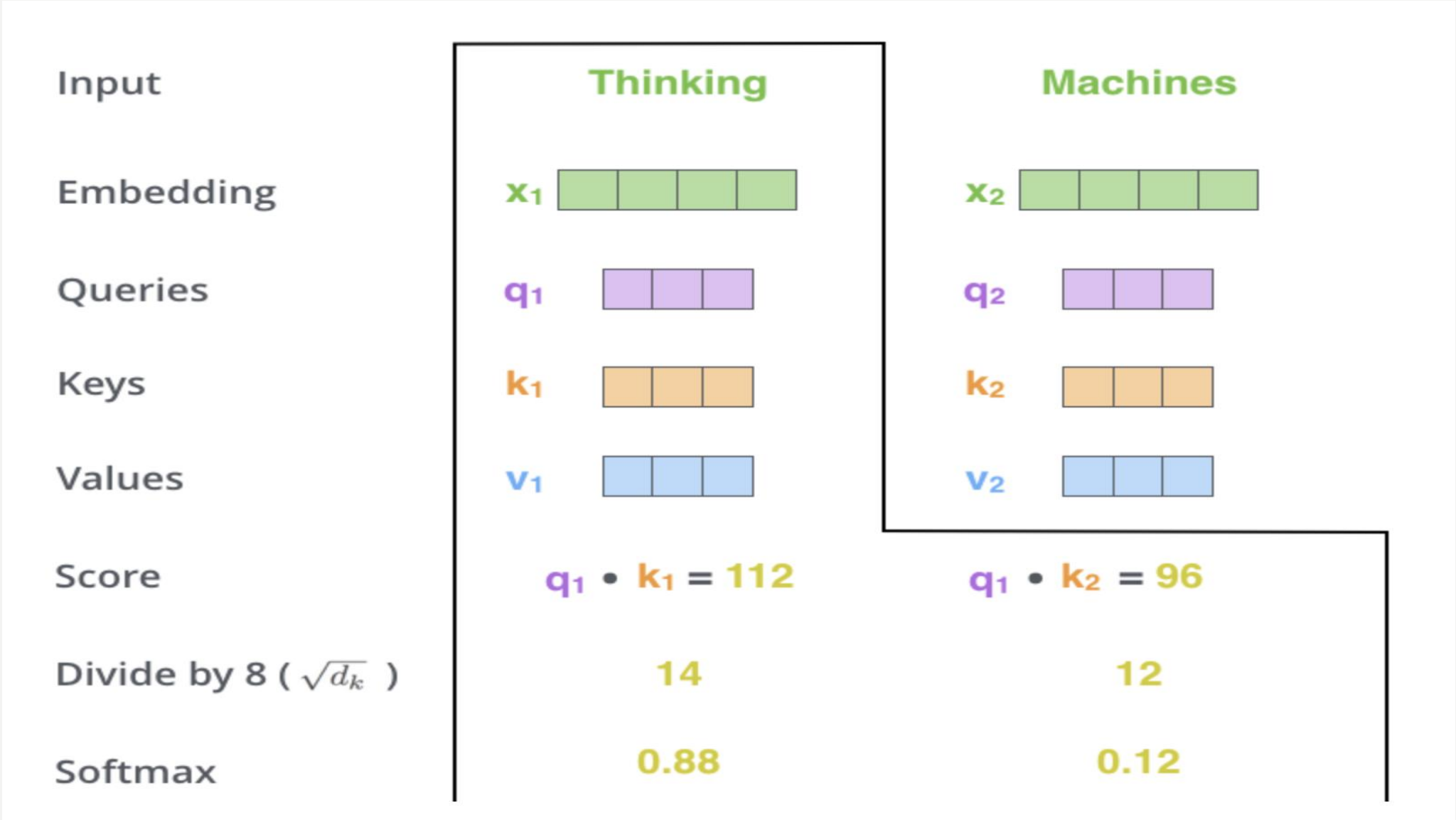


- The scaling factor $\sqrt{d_k}$ prevents large dot products that push softmax into regions with small gradients.
- **5.2 Multi-Head Attention**
- Instead of a single attention head, we compute multiple in parallel:
- $\text{head}_i = Attention\left(QW_i^Q, KW_i^K VW_i^V\right)$
- $\text{MultiHead}(Q, K, V) = Concat(head_1, \ldots, head_h)W^O$
- **Intuition:**
  Each head learns to attend to different parts of the sequence
- — syntactic, semantic, positional relationships.

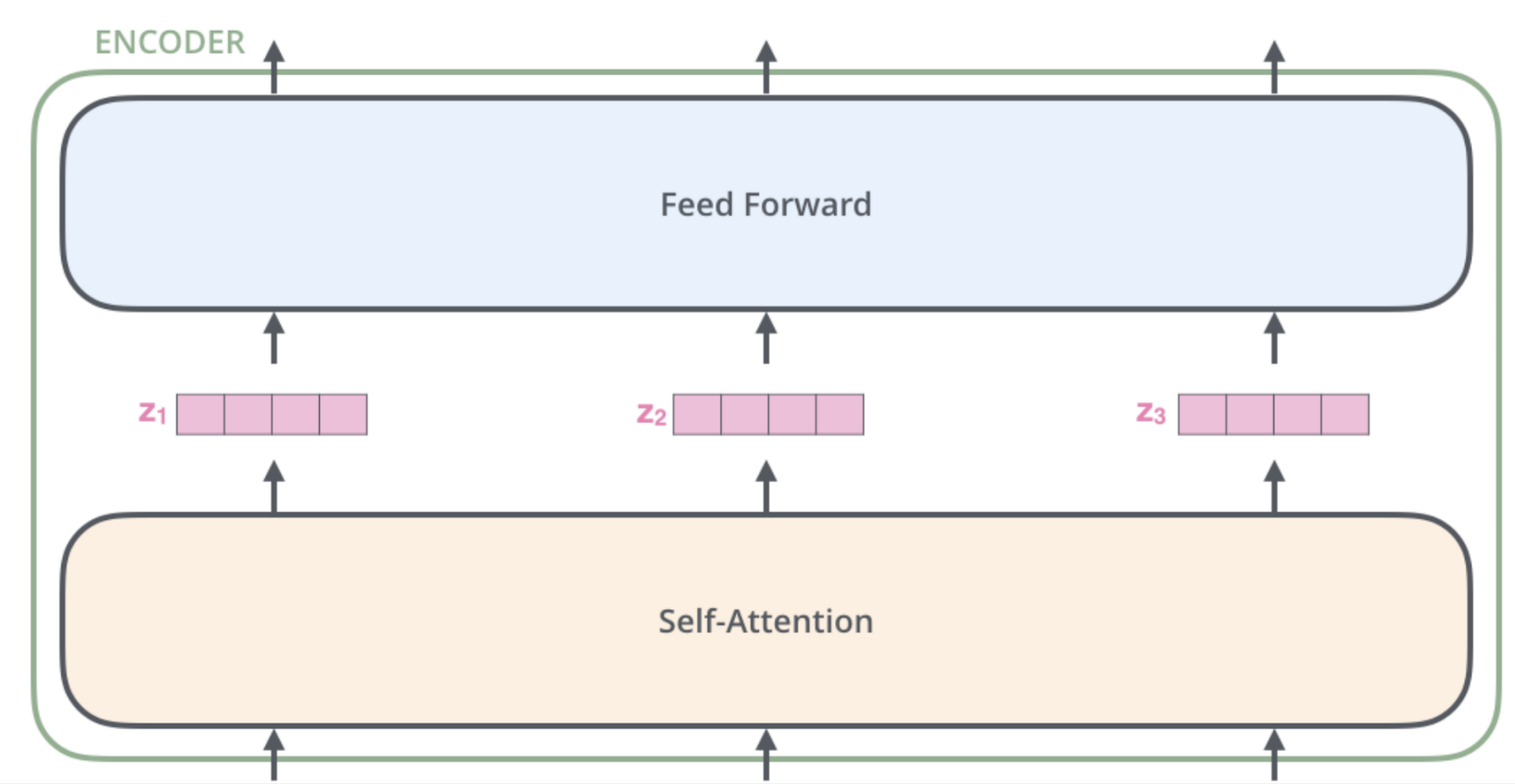




Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.



- **6. Feed-Forward Networks**
- Each layer includes a simple 2-layer fully connected network applied position-wise:
- $\mathrm{FFN}(x) = \mathit{ReLU}(xW_1 + b_1)W_2 + b_2$
- This helps project features into higher-level spaces and adds non-linearity.



- **7. Normalization and Residuals**
- Each sublayer uses:
- LayerOutput $= \mathit{LayerNorm}\big(x + \mathit{Sublayer}(x)\big)$
- Residual connections stabilize training and speed convergence.

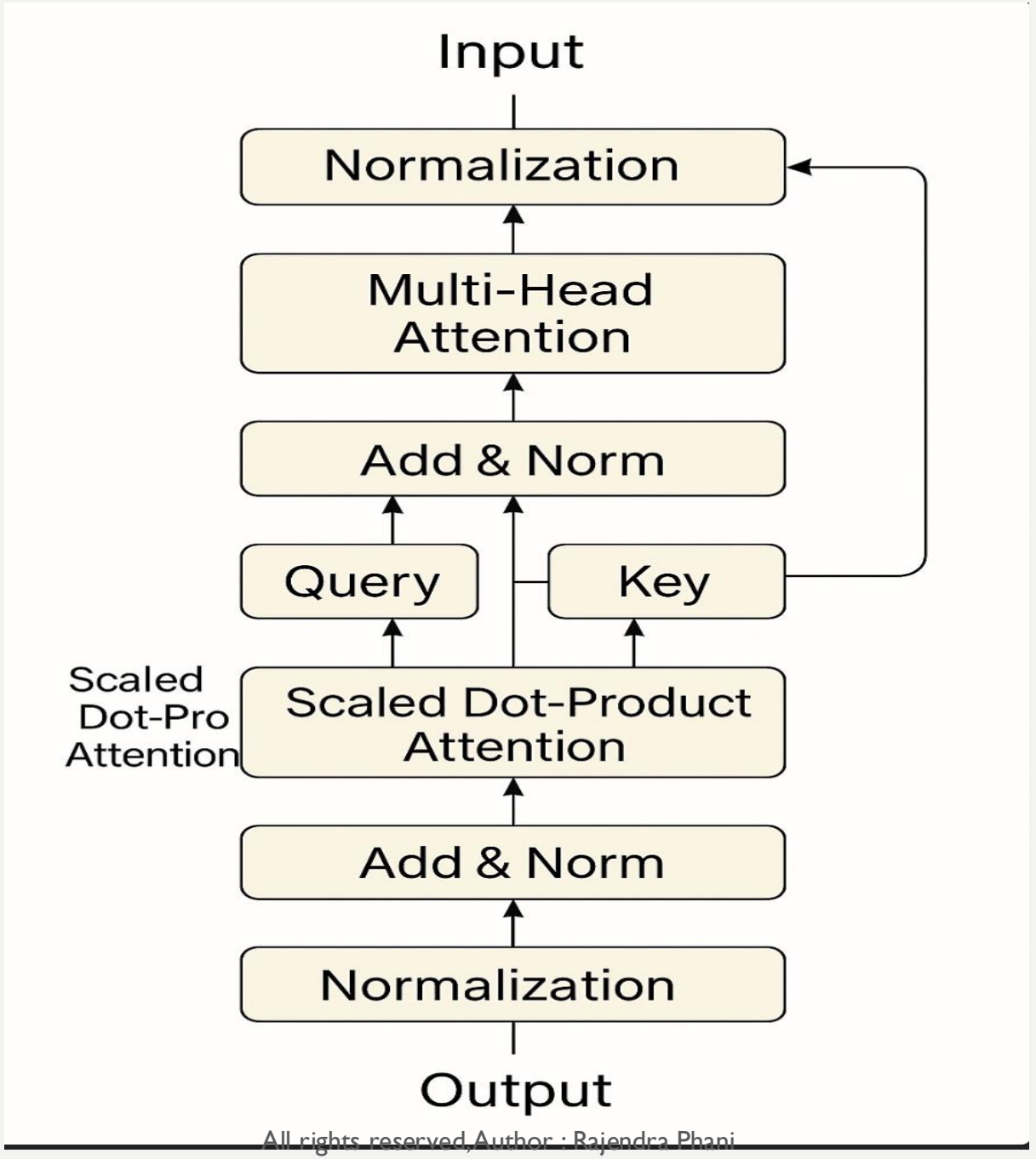
- **8. Training Process**
- **Objective:**
- **Cross-Entropy Loss** between predicted and target tokens.
- **Optimization:**
- **Adam optimizer** with custom learning rate schedule:

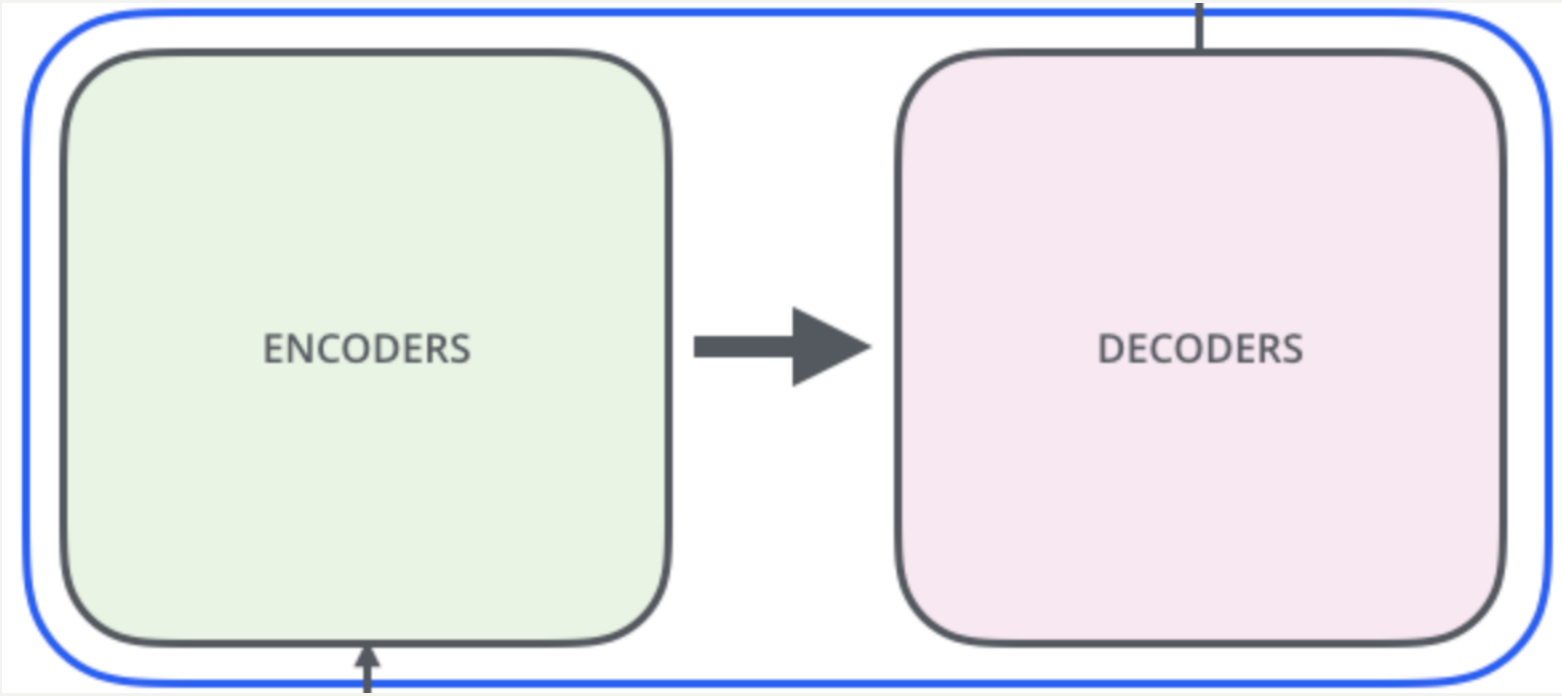
$$\text{lr} = d_{model}^{-0.5} \times \min(\text{step}^{-0.5}, \text{step} \times \text{warmup}^{-1.5})$$

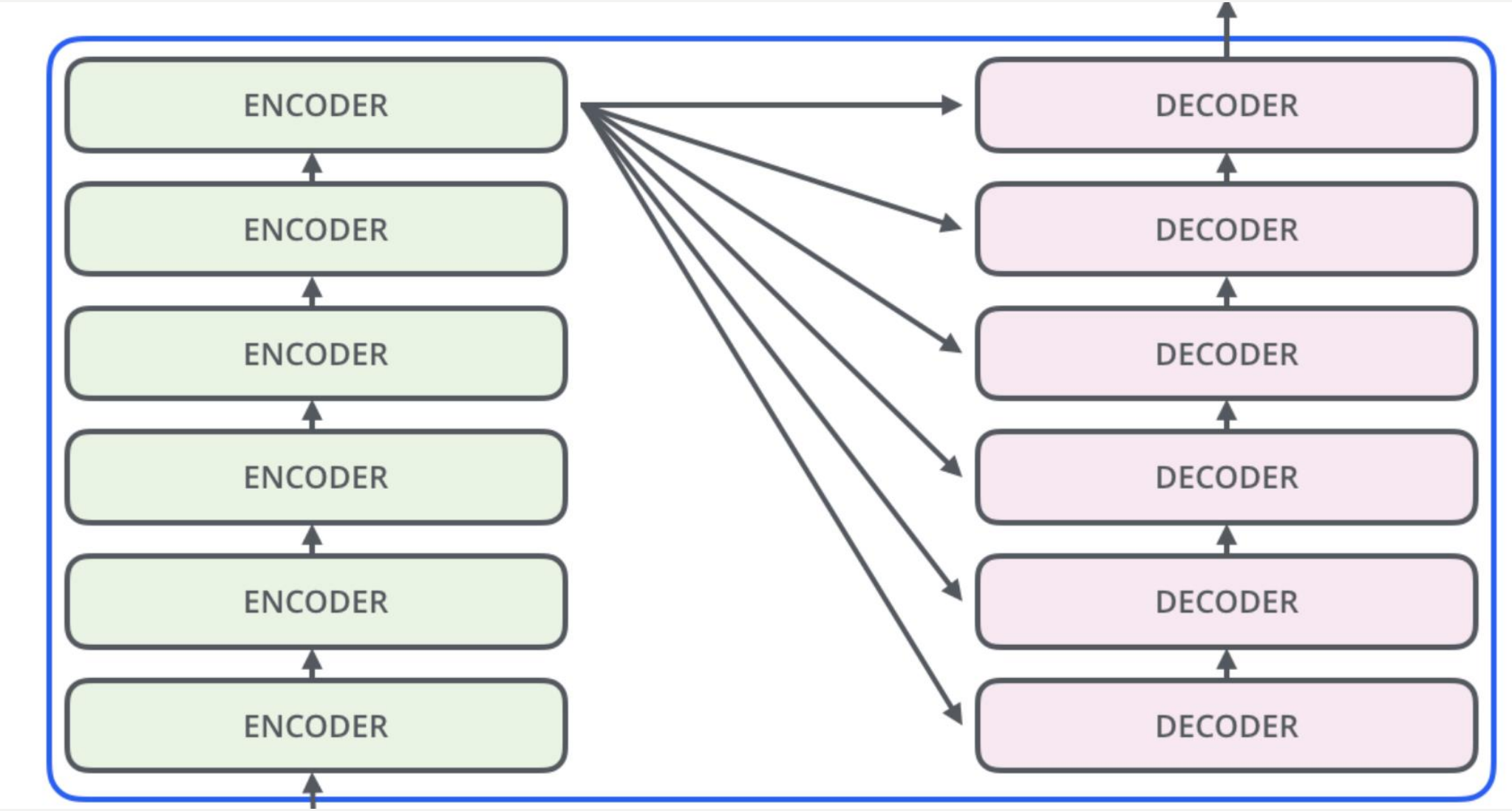
- **Regularization:**
- Dropout
- Label smoothing

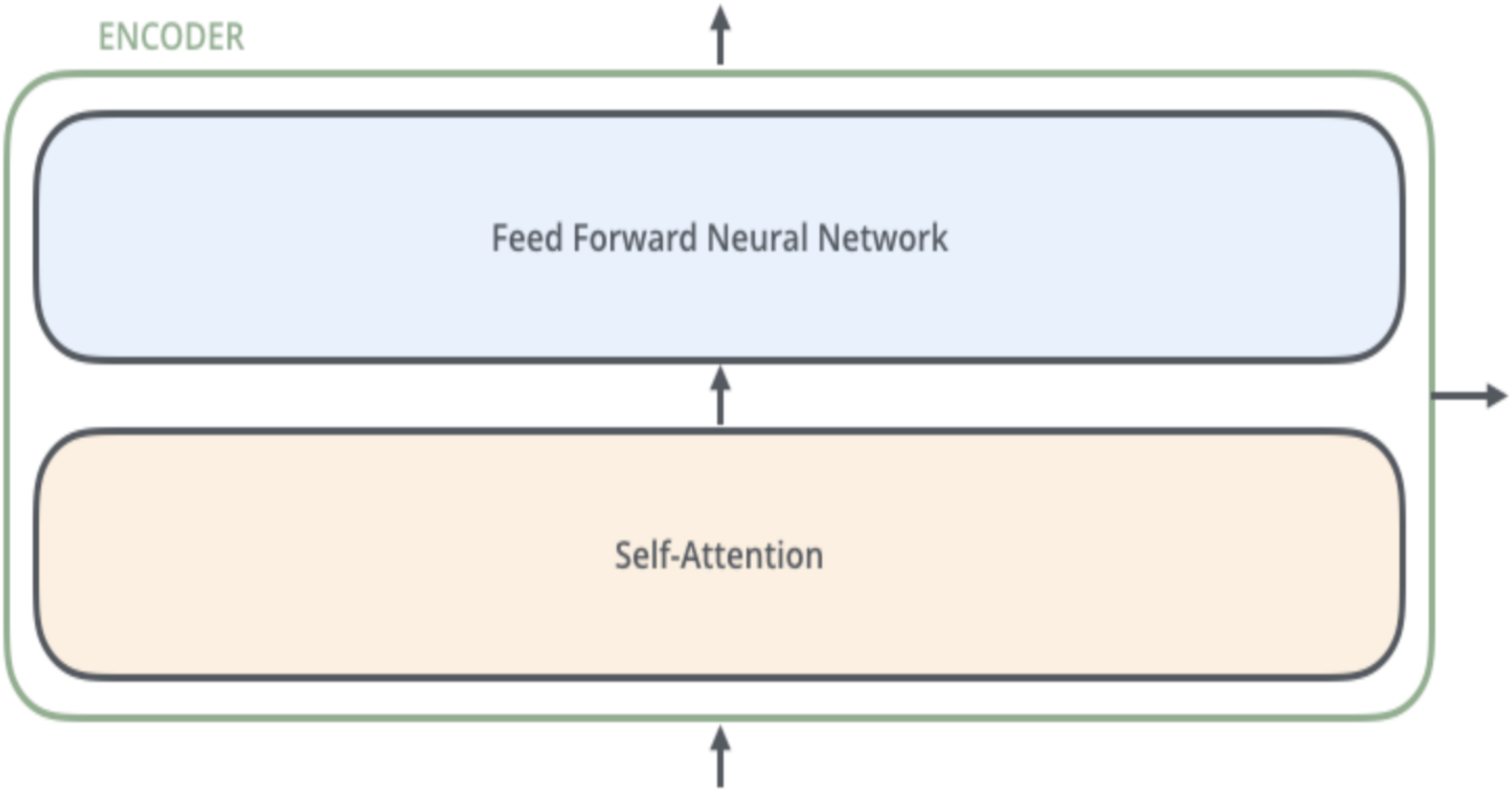
```python
import torch
import torch.nn as nn
import math

class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        assert d_model % num_heads == 0
        self.d_k = d_model // num_heads
        self.num_heads = num_heads

        self.q_linear = nn.Linear(d_model, d_model)
        self.k_linear = nn.Linear(d_model, d_model)
        self.v_linear = nn.Linear(d_model, d_model)
        self.out = nn.Linear(d_model, d_model)

    def forward(self, q, k, v, mask=None):
        B = q.size(0)
        Q = self.q_linear(q).view(B, -1, self.num_heads, self.d_k).transpose(1,2)
        K = self.k_linear(k).view(B, -1, self.num_heads, self.d_k).transpose(1,2)
        V = self.v_linear(v).view(B, -1, self.num_heads, self.d_k).transpose(1,2)

        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
        attn = torch.softmax(scores, dim=-1)
        context = torch.matmul(attn, V)

        context = context.transpose(1, 2).contiguous().view(B, -1, self.num_heads * self.
        return self.out(context)
```

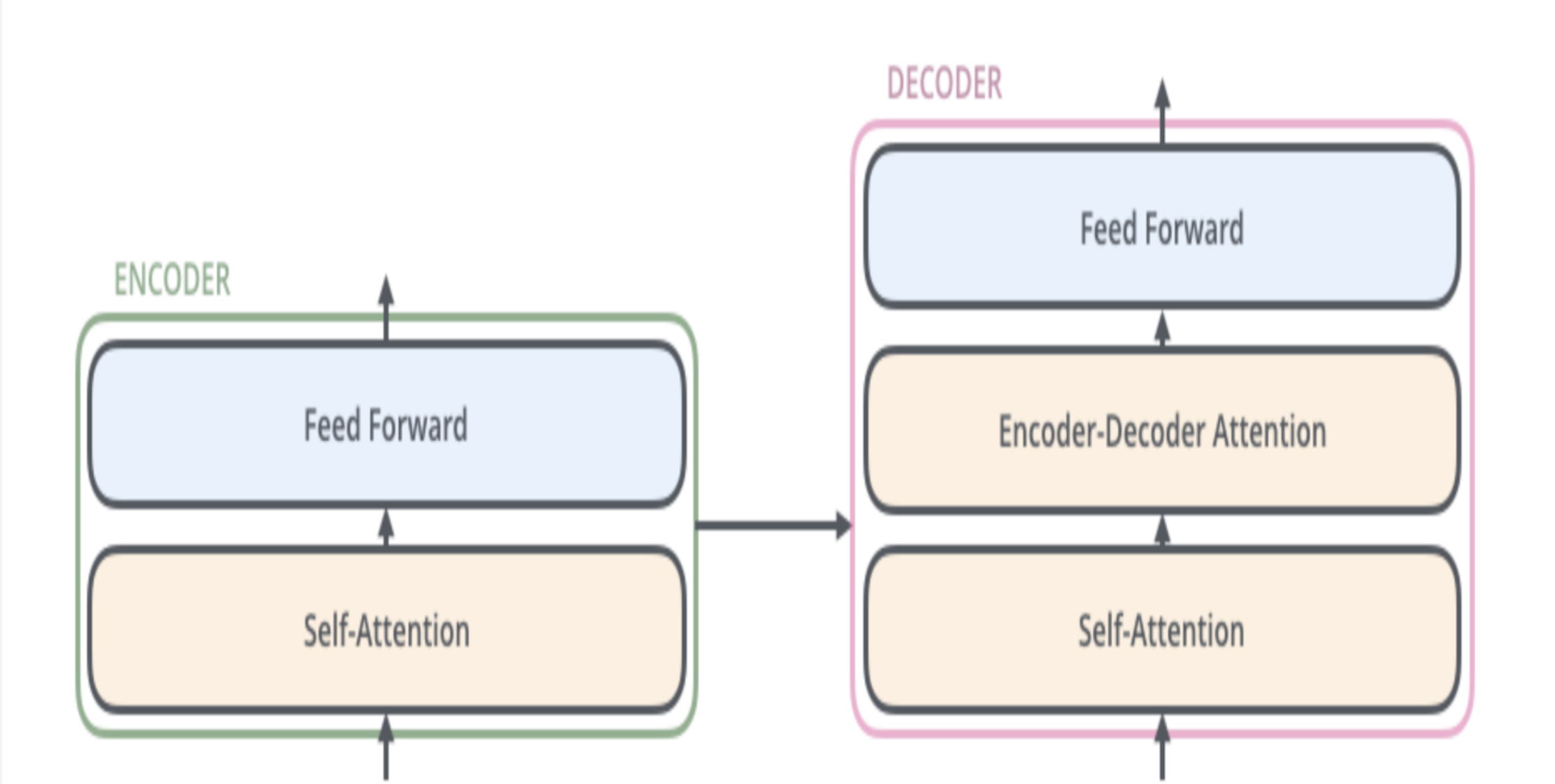
Copy code

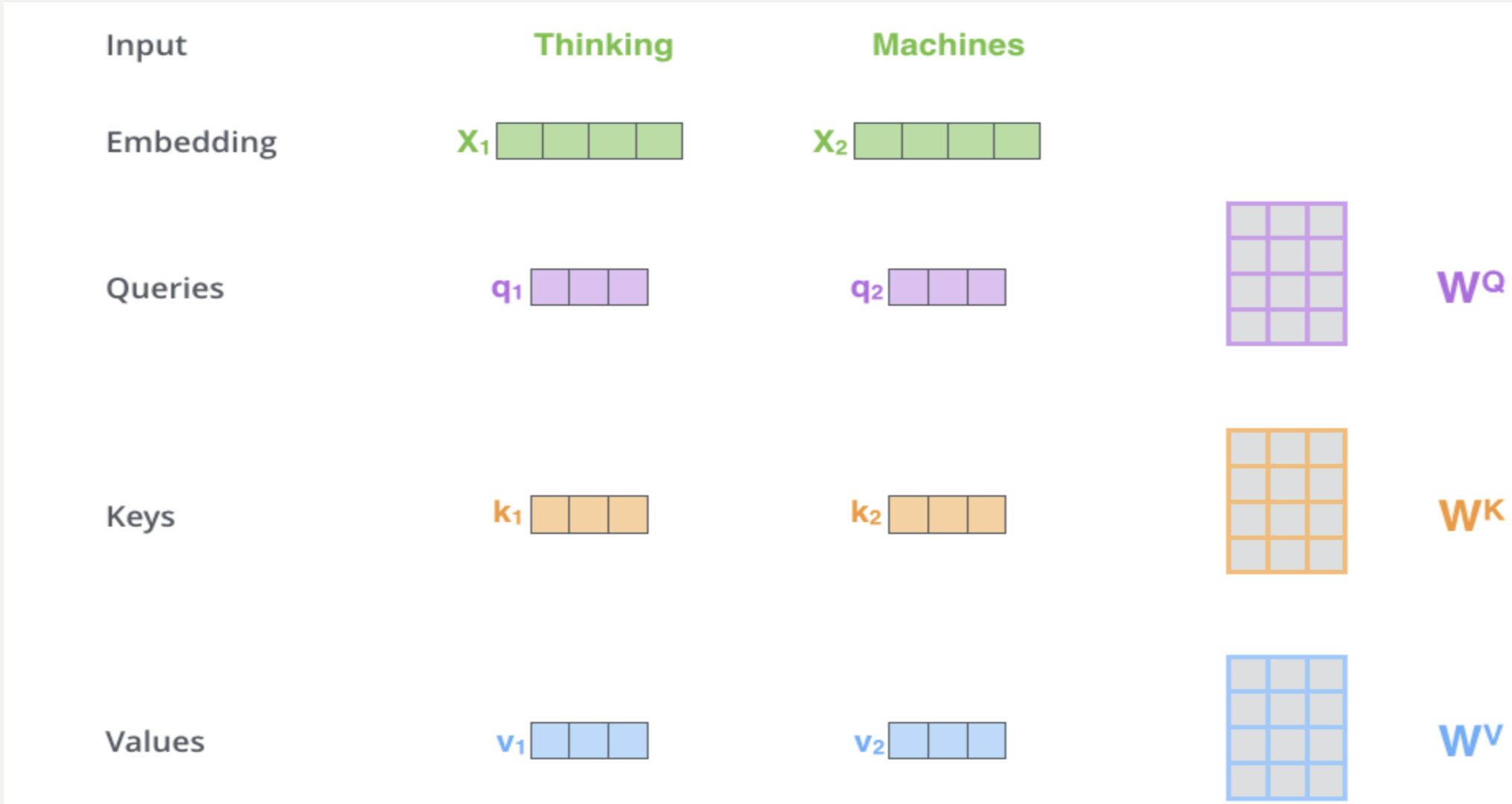




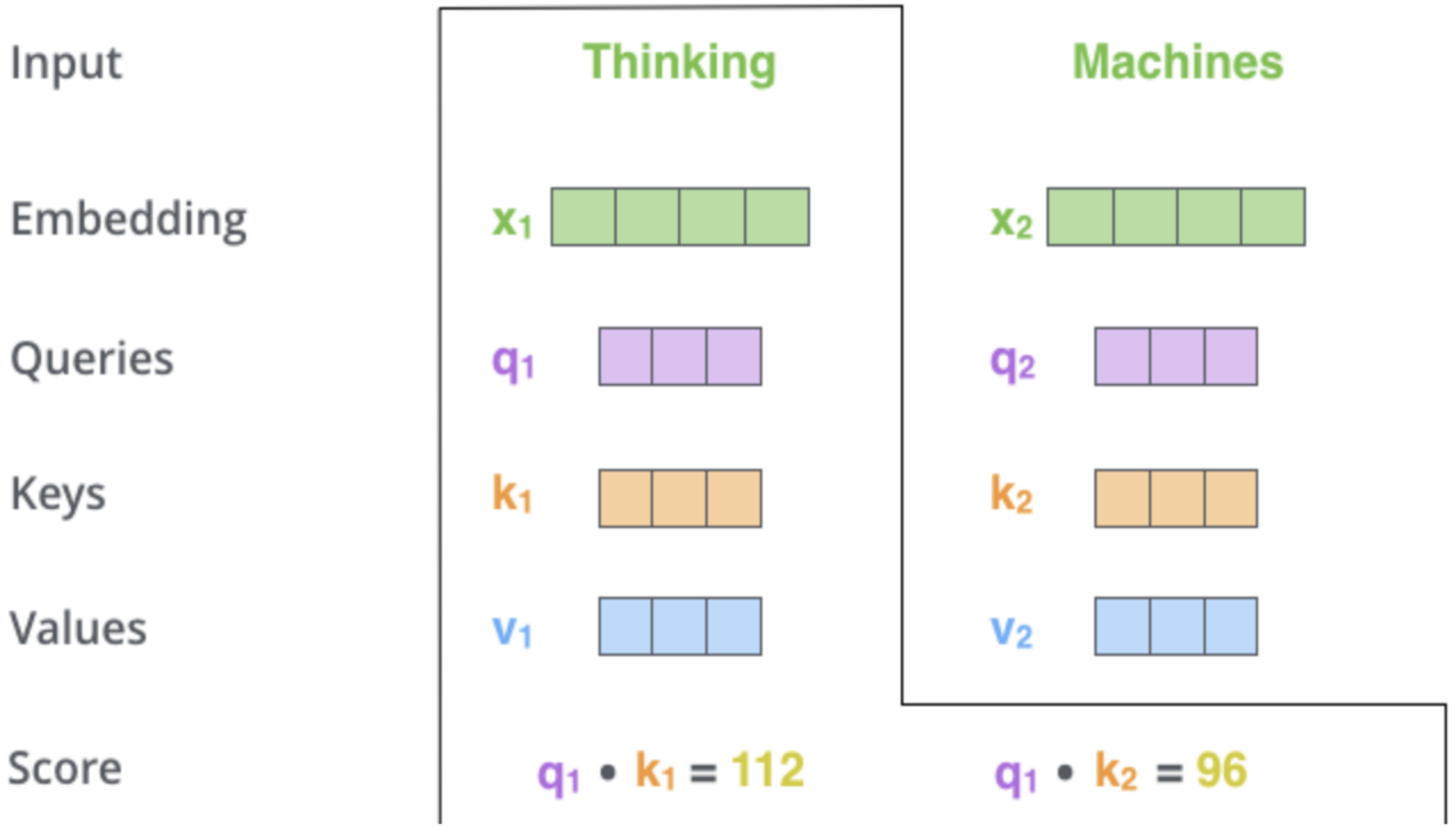


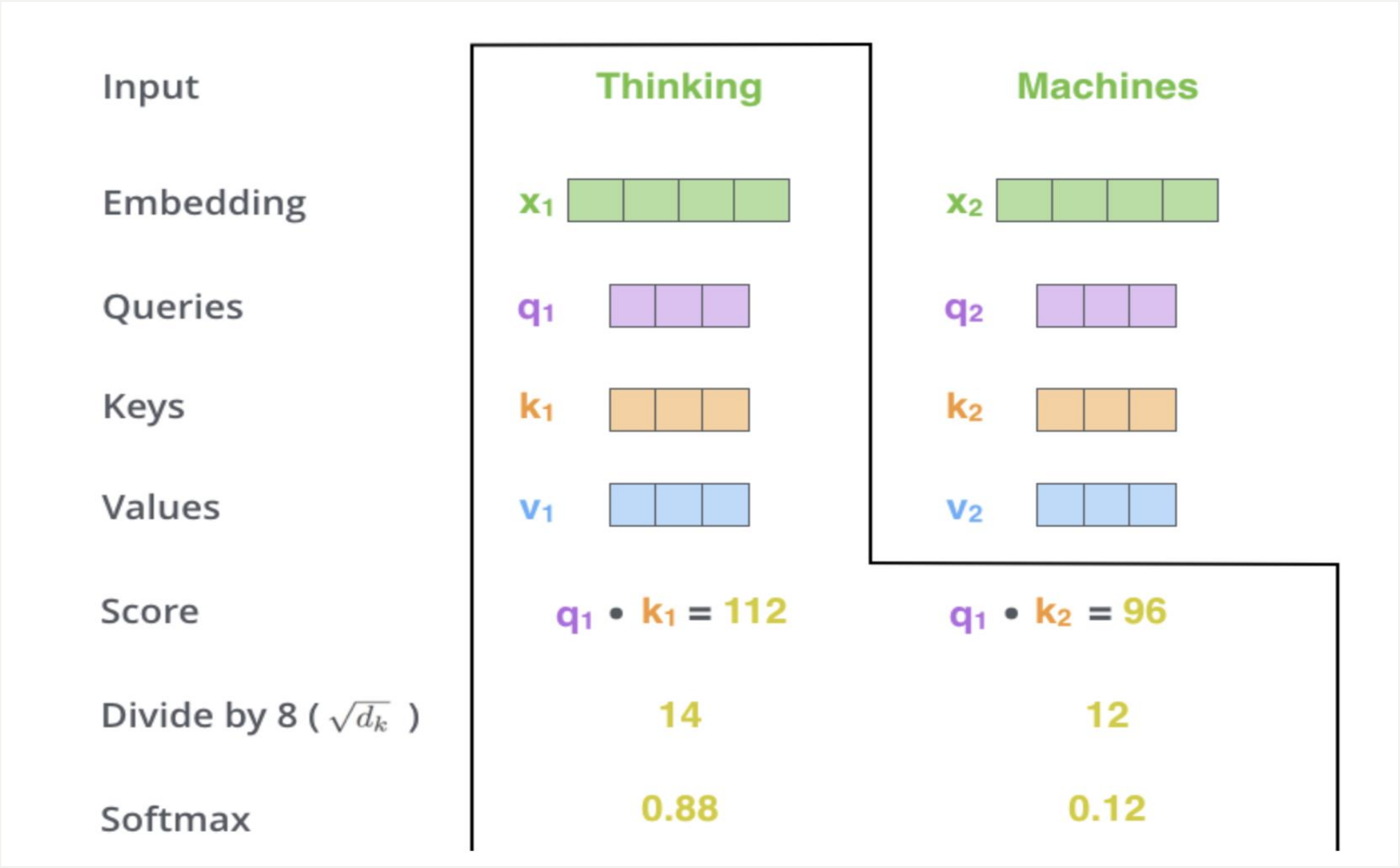


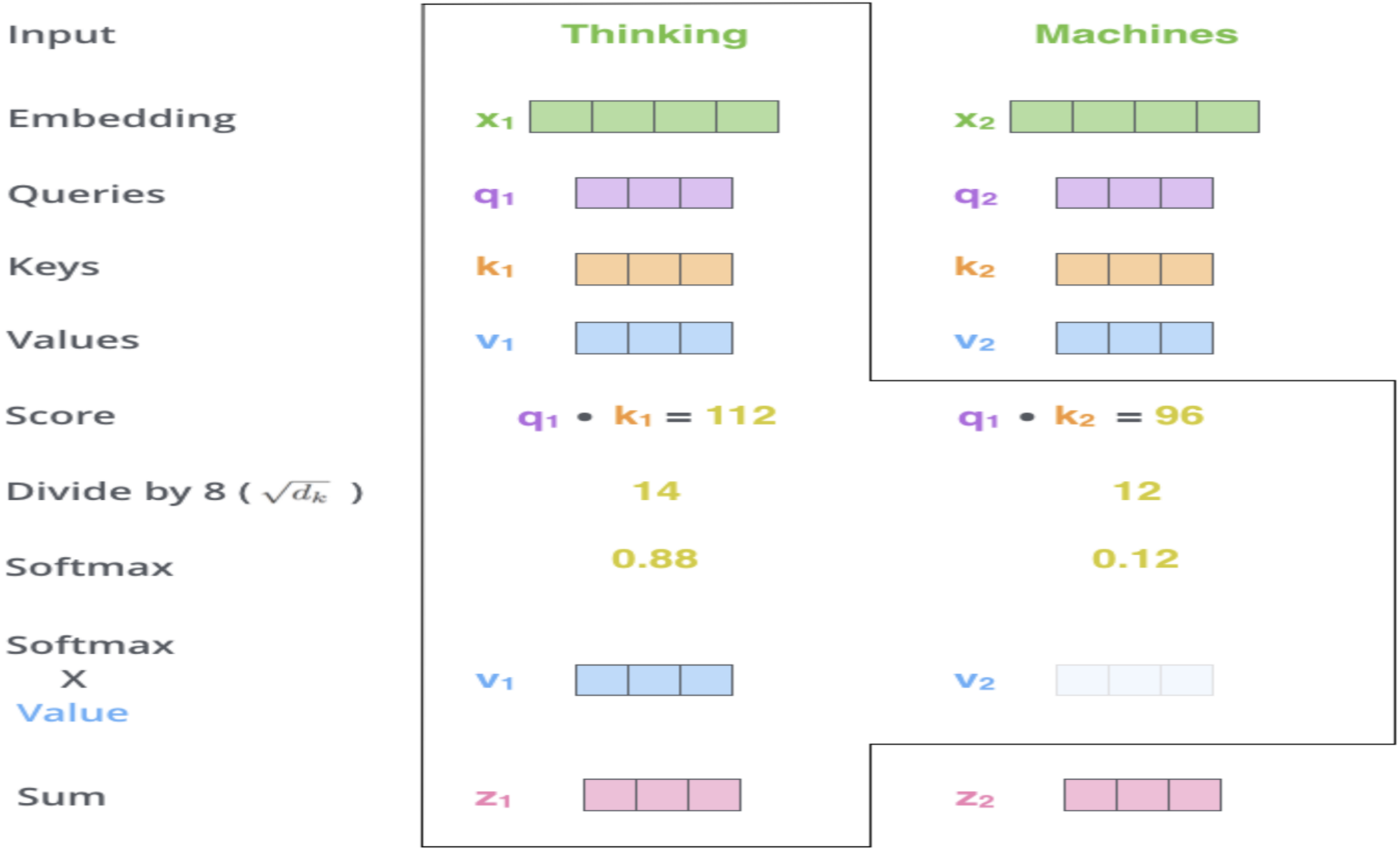


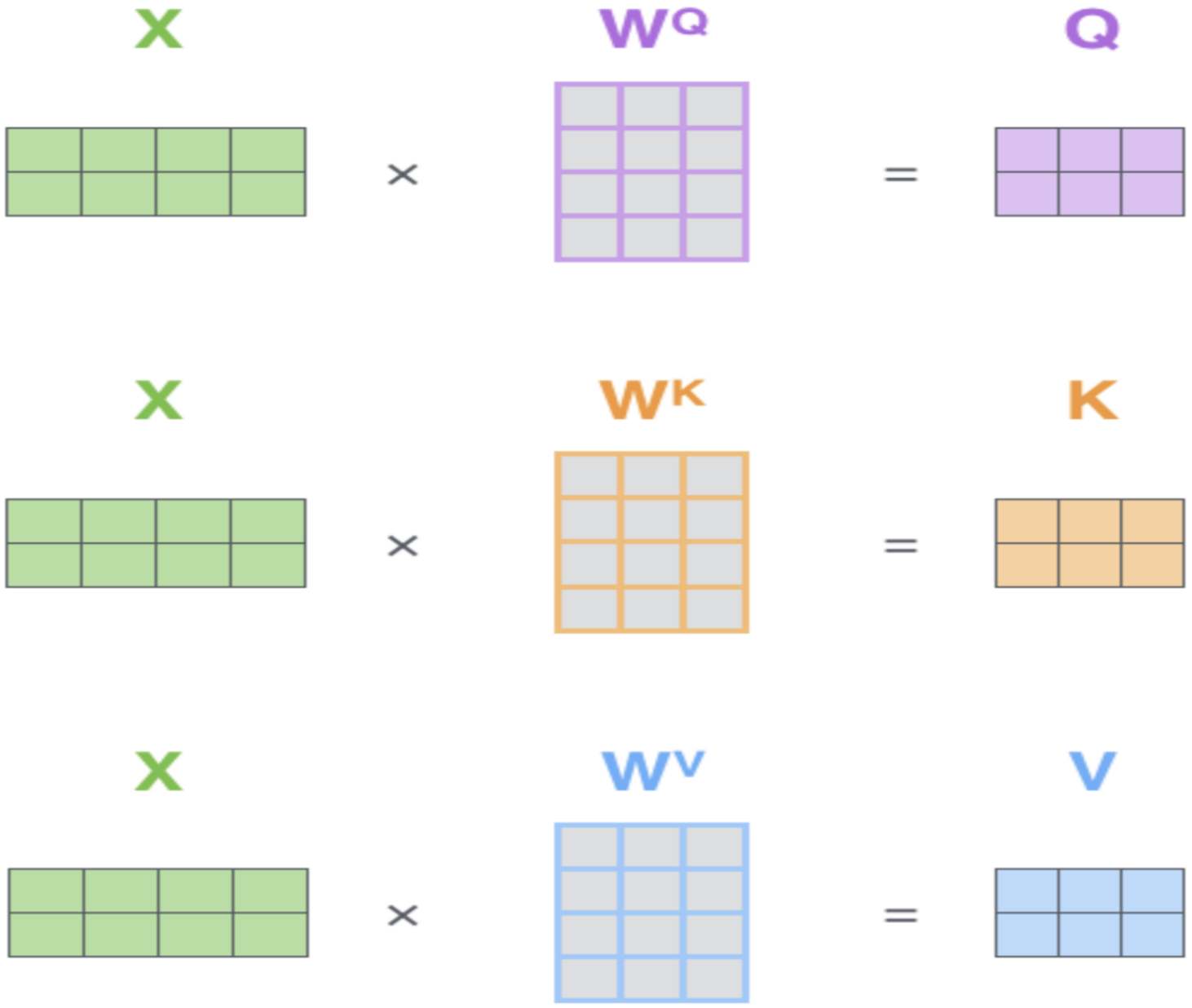


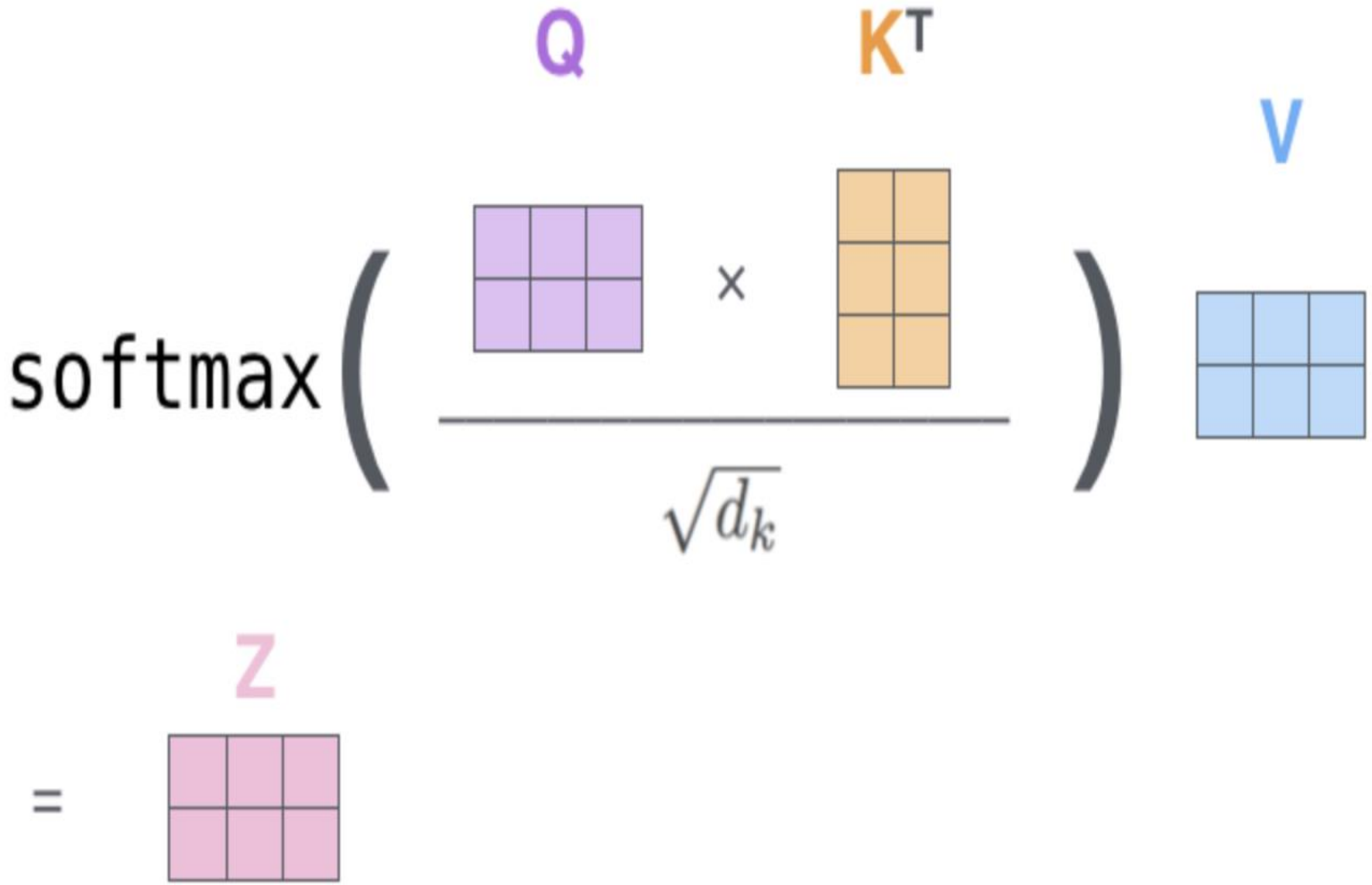
Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.











1) This is our input sentence*

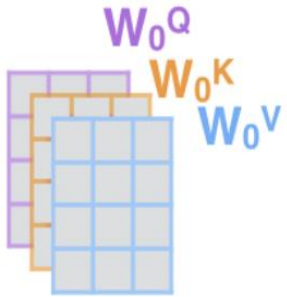
2) We embed each word*

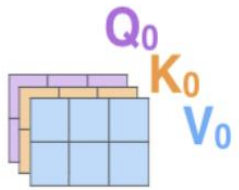
3) Split into 8 heads. We multiply X or R with weight matrices

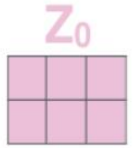
4) Calculate attention using the resulting Q/K/V matrices

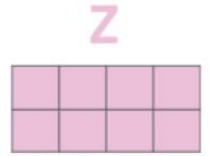
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

Thinking Machines

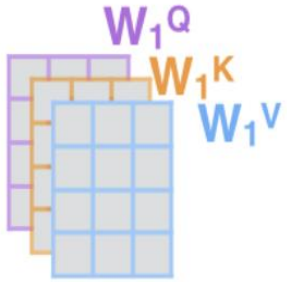


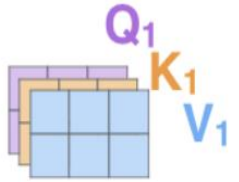


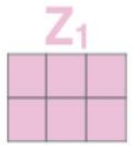


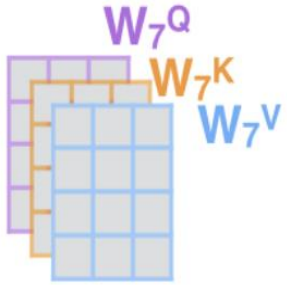


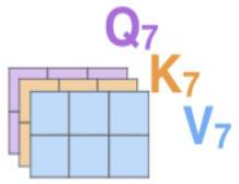
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

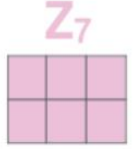


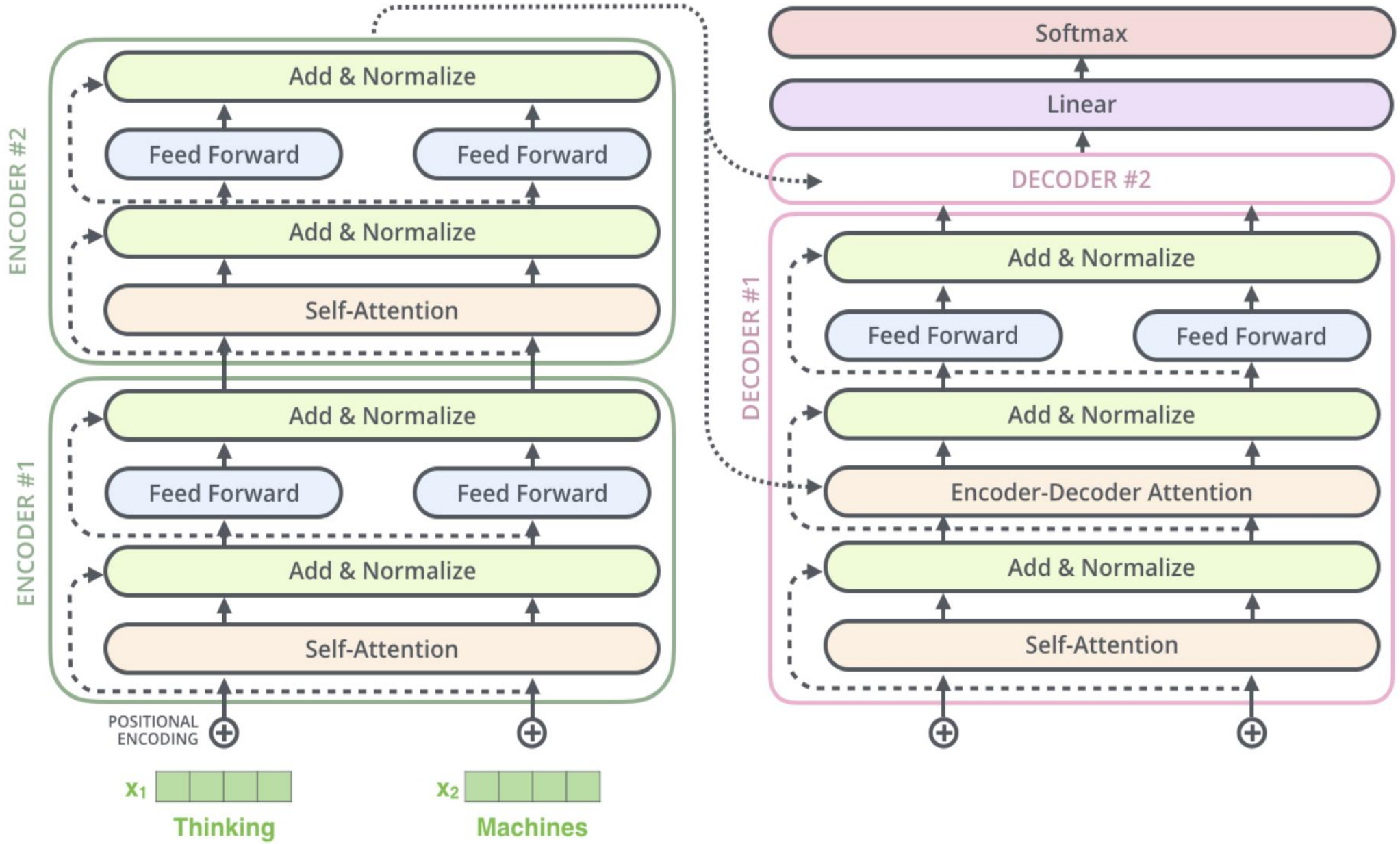


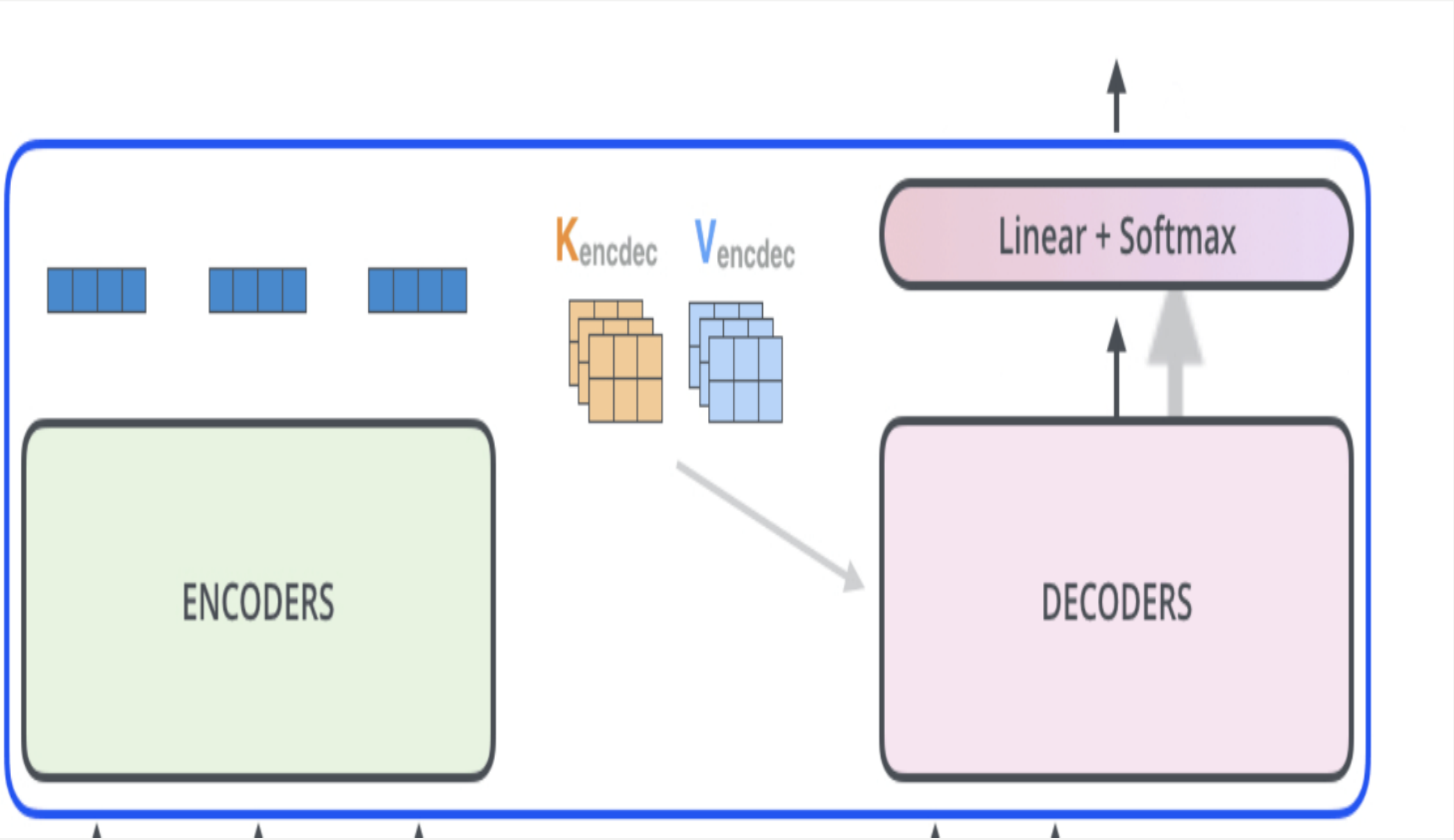












- **1) Input embedding + positional encoding**
- **Why**: tokens need vector representations; model has no recurrence so we add position signal.
- **Math**:
- token embedding: E = Embedding(x)
- sinusoidal positional encoding (original Vaswani):
  - PE(pos, 2i) = sin(pos / 10000^{2i/d})
  - PE(pos, 2i+1) = cos(pos / 10000^{2i/d})
- input: X0 = E + PE
- **Bullets (stepwise)**:
- Map token ids → d-dimensional vectors.
- Compute (or learn) positional encodings.
- Add positional encodings elementwise to embeddings.
- Apply dropout (optional).

python

Copy code

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

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)  # (max_len, d)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)  # (max_len,1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)  # (1, max_len, d_model)
        self.register_buffer('pe', pe)  # not a parameter
    def forward(self, x):
        # x: (B, T, d_model)
        return x + self.pe[:, :x.size(1)]
```

- **2) Scaled Dot-Product Attention (single head)**
- **Why**: compute weighted sum of values based on similarity between queries and keys.
- **Math**:
- $\text{Attention}(Q, K, V) = \textit{softmax}\left(\frac{QK^{\top}}{\sqrt{d_k}}\right)V$
- Q ∈ ℝ^{B×T_q×d_k}, K,V ∈ ℝ^{B×T_k×d_k}
- Masking: add -inf to logits where mask==0 before softmax.
- **Bullets**:
- Compute dot-products between Q and $K^{\top}$ → raw scores.
- Scale by 1/√d_k.
- Optionally apply mask (prevent attending to future tokens or padding).
- Softmax across key positions → attention weights.
- Weighted sum of V → context.

python

Copy code

```python
def scaled_dot_product_attention(q, k, v, mask=None, dropout=None):
    # q, k, v: (B, h, T, d_k) if multihead split; but works for head-specific shapes
    d_k = q.size(-1)
    scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(d_k)  # (B,h,T_q,T_k)
    if mask is not None:
        # mask shape should broadcast to scores
        scores = scores.masked_fill(mask == 0, float('-inf'))
    attn = torch.softmax(scores, dim=-1)
    if dropout is not None:
        attn = dropout(attn)
    output = torch.matmul(attn, v)  # (B,h,T_q,d_k)
    return output, attn
```

- **3) Multi-Head Attention**
- **Why**: multiple attention "heads" let the model attend to different subspaces / relations.
- **Math**:
- For head i:
  - $Q\_i = Q W\_i^Q, K\_i = K W\_i^K, V\_i = V W\_i^V$ where $W\_i^* \in \mathbb{R}^{d\_model \times d\_k}$
  - $head\_i = Attention(Q\_i, K\_i, V\_i)$
- Concatenate heads and project: $MultiHead(Q,K,V) = Concat(head\_1..h)\, W^O$
- **Bullets**:
- Linearly project Q/K/V into h smaller subspaces.
- Run scaled dot-product attention in parallel for each head.
- Concatenate results and linearly project back to d_model.

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, n_heads, dropout=0.1):
        super().__init__()
        assert d_model % n_heads == 0
        self.d_model = d_model
        self.n_heads = n_heads
        self.d_k = d_model // n_heads

        self.w_q = nn.Linear(d_model, d_model)
        self.w_k = nn.Linear(d_model, d_model)
        self.w_v = nn.Linear(d_model, d_model)
        self.w_o = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, q, k, v, mask=None):
        # q,k,v: (B,T, d_model)
        B = q.size(0)
        def shape(x):
            # -> (B, n_heads, T, d_k)
            return x.view(B, -1, self.n_heads, self.d_k).transpose(1, 2)
        Q = shape(self.w_q(q))
        K = shape(self.w_k(k))
        V = shape(self.w_v(v))

        if mask is not None:
            # mask expected shape: (B, 1, T_q, T_k) or (B, n_heads, T_q, T_k)
            mask = mask.unsqueeze(1) if mask.dim() == 3 else mask

        x, attn = scaled_dot_product_attention(Q, K, V, mask=mask, dropout=self.dropout)
        # x: (B, n_heads, T_q, d_k)
        x = x.transpose(1, 2).contiguous().view(B, -1, self.n_heads * self.d_k)  # (B, T_q,
        return self.w_o(x), attn  # (B, T_q, d_model)
```

Copy code

- **4) Position-wise Feed-Forward Network (FFN)**
- **Why**: add non-linear projection independently to each position.
- **Math**:
- Applied identically and independently to every position.
- **Bullets**:
- Two linear layers with ReLU (or GELU), applied per position.
- Usually d_ff ≈ 4 * d_model in original paper.

```python
class PositionwiseFFN(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model)
        )
    def forward(self, x):
        return self.net(x)  # (B, T, d_model)
```

- **5) Layer normalization + Residual connections**
- **Why**: residuals ease gradient flow; layernorm stabilizes training.
- **Operation** (sublayer connection):
- SublayerOutput $= \mathit{LayerNorm}\big(x + \mathit{Sublayer}(x)\big)$
- **Bullets**:
- After each sublayer (attention or FFN), add input (residual) then apply LayerNorm.
- Apply dropout on the sublayer output before adding residual (implementation choice).
- **Code pattern**:

```python
class SublayerConnection(nn.Module):
    def __init__(self, d_model, dropout=0.1):
        super().__init__()
        self.norm = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x, sublayer_fn):
        # sublayer_fn: function that returns sublayer output given x
        return x + self.dropout(sublayer_fn(self.norm(x)))
```

- **) Encoder layer & Encoder stack**
- **Encoder layer structure (per layer)**:
- Multi-Head Self-Attention (Q=K=V from layer input)
  - Residual + LayerNorm
- Position-wise FFN
  - Residual + LayerNorm
- **Bullets**:
- Self-attention allows each position to attend to all positions in the same input.
- Stack N identical layers (N=6 typical).
- **Code**:

```python
class EncoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, d_ff, dropout=0.1):
        s
        self.self_attn = MultiHeadAttention(d_model, n_heads, dropout)
        self.ffn = PositionwiseFFN(d_model, d_ff, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, src_mask=None):
        # x: (B, T, d_model)
        # Self-attn
        attn_out, attn = self.self_attn(x, x, x, mask=src_mask)
        x = x + self.dropout(attn_out)
        x = self.norm1(x)
        # FFN
        ff_out = self.ffn(x)
        x = x + self.dropout(ff_out)
        x = self.norm2(x)
        return x, attn

class Encoder(nn.Module):
    def __init__(self, layer, N):
        super().__init__()
        self.layers = nn.ModuleList([layer for _ in range(N)])
    def forward(self, x, src_mask=None):
        attns = []
        for layer in self.layers:
            x, attn = layer(x, src_mask)
            attns.append(attn)
        return x, attns
```

- **7) Decoder layer & Decoder stack**
- **Decoder layer structure (per layer)**:
- Masked Multi-Head Self-Attention (prevents looking at future tokens)
  - Residual + LayerNorm
- Encoder-Decoder Multi-Head Attention (Q from decoder, K,V from encoder outputs)
  - Residual + LayerNorm
- Position-wise FFN
  - Residual + LayerNorm
- **Bullets**:
- Masked self-attention ensures autoregressive property: at decoding step t, model cannot access tokens > t.
- Encoder-decoder attention allows decoder to attend to encoder output for cross-attention (e.g., source sentence).
- Stack N identical decoder layers.

```python
class DecoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, d_ff, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, n_heads, dropout)
        self.enc_dec_attn = MultiHeadAttention(d_model, n_heads, dropout)
        self.ffn = PositionwiseFFN(d_model, d_ff, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc_out, tgt_mask=None, memory_mask=None):
        # 1) Masked self-attention
        attn_out, self_attn_map = self.self_attn(x, x, x, mask=tgt_mask)
        x = x + self.dropout(attn_out); x = self.norm1(x)
        # 2) Encoder-decoder attention
        attn_out, encdec_attn_map = self.enc_dec_attn(x, enc_out, enc_out, mask=memory_ma
        x = x + self.dropout(attn_out); x = self.norm2(x)
        # 3) Feed-forward
        ff_out = self.ffn(x)
        x = x + self.dropout(ff_out); x = self.norm3(x)
        return x, self_attn_map, encdec_attn_map

class Decoder(nn.Module):
    def __init__(self, layer, N):
        super().__init__()
        self.layers = nn.ModuleList([layer for _ in range(N)])
    def forward(self, x, enc_out, tgt_mask=None, memory_mask=None):
        self_attns, encdec_attns = [], []
        for layer in self.layers:
            x, self_attn, encdec_attn = layer(x, enc_out, tgt_mask, memory_mask)
            self_attns.append(self_attn)
            encdec_attns.append(encdec_att
        return x, self_attns, encdec_attns
```

- **8) Masking (practical details)**
- **Types**:
- **Padding mask**: mask out padded tokens (both encoder and decoder attend to non-padding).
- **Subsequent (look-ahead) mask**: triangular mask for decoder self-attention to prevent attending to future positions.
- **Code helpers**:

```python
def make_pad_mask(seq, pad_idx=0):
    # seq: (B, T)
    return (seq != pad_idx).unsqueeze(1).unsqueeze(2)  # (B,1,1,T) broadcastable


def make_subsequent_mask(size):
    # returns (1, 1, size, size) or (size, size)
    mask = torch.triu(torch.ones(size, size), diagonal=1).bool()
    return ~mask  # True where allowed
```

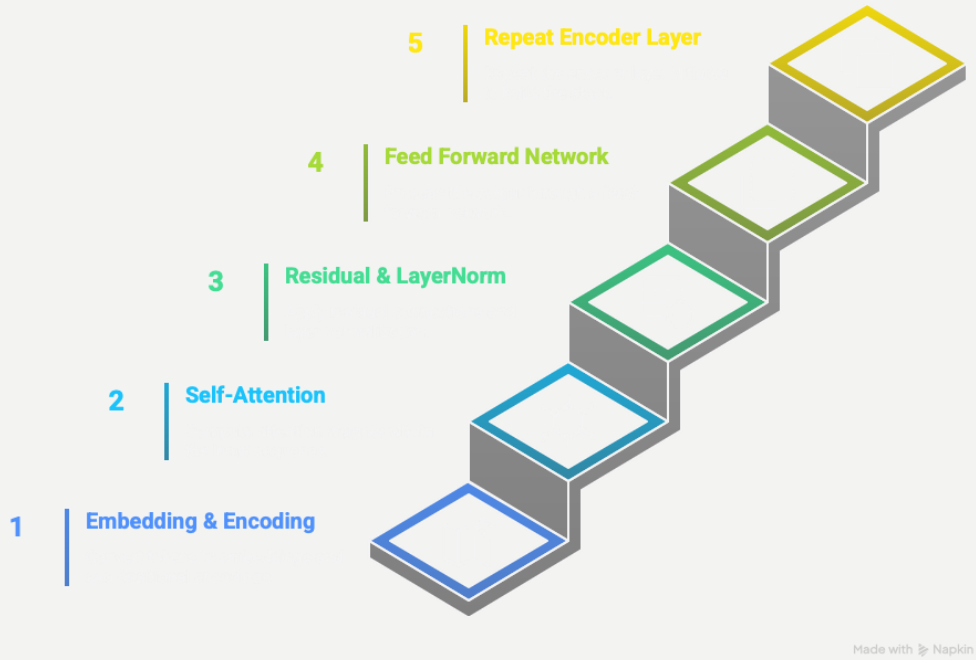
- **9) Complete Transformer model & final linear layer**
- **Bullets**:
- Build Embedding + PosEnc for source and target vocab.
- Run encoder: encoder_out = Encoder(X0)
- Run decoder: decoder_out = Decoder(Y0, encoder_out)
- Final Linear(d_model -> vocab_size) to produce logits.

```python
class TransformerModel(nn.Module):
    def __init__(self, src_vocab, tgt_vocab, d_model=512, N=6, n_heads=8, d_ff=2048, dropou
        super().__init__()
        self.src_embed = nn.Embedding(src_vocab, d_model)
        self.tgt_embed = nn.Embedding(tgt_vocab, d_model)
        self.pos_enc = PositionalEncoding(d_model, max_len)
        layer_enc = EncoderLayer(d_model, n_heads, d_ff, dropout)
        self.encoder = Encoder(layer_enc, N)
        layer_dec = DecoderLayer(d_model, n_heads, d_ff, dropout)
        self.decoder = Decoder(layer_dec, N)
        self.out = nn.Linear(d_model, tgt_vocab)

    def forward(self, src, tgt, src_mask=None, tgt_mask=None, memory_mask=None):
        # src: (B, T_src), tgt: (B, T_tgt)
        src_emb = self.pos_enc(self.src_embed(src))
        tgt_emb = self.pos_enc(self.tgt_embed(tgt))
        enc_out, enc_attns = self.encoder(src_emb, src_mask)
        dec_out, self_attns, encdec_attns = self.decoder(tgt_emb, enc_out, tgt_mask, memory
        logits = self.out(dec_out)  # (B, T_tgt, vocab)
        return logits, enc_attns, self_attns, encdec_attns
```

- **10) Loss, optimization, and LR schedule**
- **Loss**: cross-entropy on logits (shifted decoder target) with optional label smoothing.
- **Optimizer**: Adam (often with betas (0.9, 0.98)).
- **LR schedule** (Vaswani):

```
model = TransformerModel(src_vocab=10000, tgt_vocab=10000)  # example
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, betas=(0.9, 0.98), eps=1e-9)

criterion = nn.CrossEntropyLoss(ignore_index=0)  # assume 0 is pad

# toy training step:
src = torch.randint(1, 9999, (32, 20))  # (B, T_src)
tgt = torch.randint(1, 9999, (32, 10))  # (B, T_tgt)
tgt_input = tgt[:, :-1]  # decoder input (shifted right)
tgt_labels = tgt[:, 1:]  # decoder targets

src_mask = make_pad_mask(src)  # broadcastable
tgt_mask = make_pad_mask(tgt_input) & make_subsequent_mask(tgt_input.size(1)).to(src.devi

logits, *_ = model(src, tgt_input, src_mask=src_mask, tgt_mask=tgt_mask)
# logits: (B, T_tgt-1, vocab)
loss = criterion(logits.view(-1, logits.size(-1)), tgt_labels.contiguous().view(-1))
loss.backward()
optimizer.step()
```

- Per-layer checklist (bullet list you can use for documentation pages)
- **Embedding + PositionalEncoding**
- Token embedding lookup (B, T) -> (B, T, d_model)
- Compute positional encodings (1, T, d_model)
- Add encodings and dropout
- **Encoder Layer**
- Self-attention: Q=K=V=input
- Apply padding mask (if any)
- Residual & LayerNorm
- FFN (d_model → d_ff → d_model)
- Residual & LayerNorm
- **Encoder Stack**
- Repeat encoder layer N times
- Optionally collect attention maps for diagnostics



- **Decoder Layer**
- Masked self-attention: prevent future tokens
- Residual & LayerNorm
- Encoder-decoder attention: Q=decoder_out, K,V=encoder_out
- Residual & LayerNorm
- FFN and final Residual & LayerNorm
- **Decoder Stack**
- Repeat decoder layer N times
- Final linear projection to vocab
- **Training**
- Prepare masks (padding + subsequent)
- Construct decoder input by shifting target right (<s> t_0 t_1 ...)
- Compute logits -> cross-entropy with target (shifted left)
- Use Adam + LR schedule + label smoothing/dropout