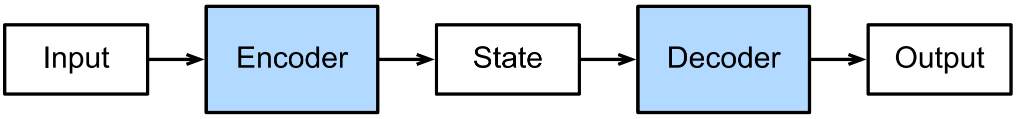
From Encoder-Decoder To Transformer

> images are from https://zh-v2.d2l.ai/

Encoder-Decoder

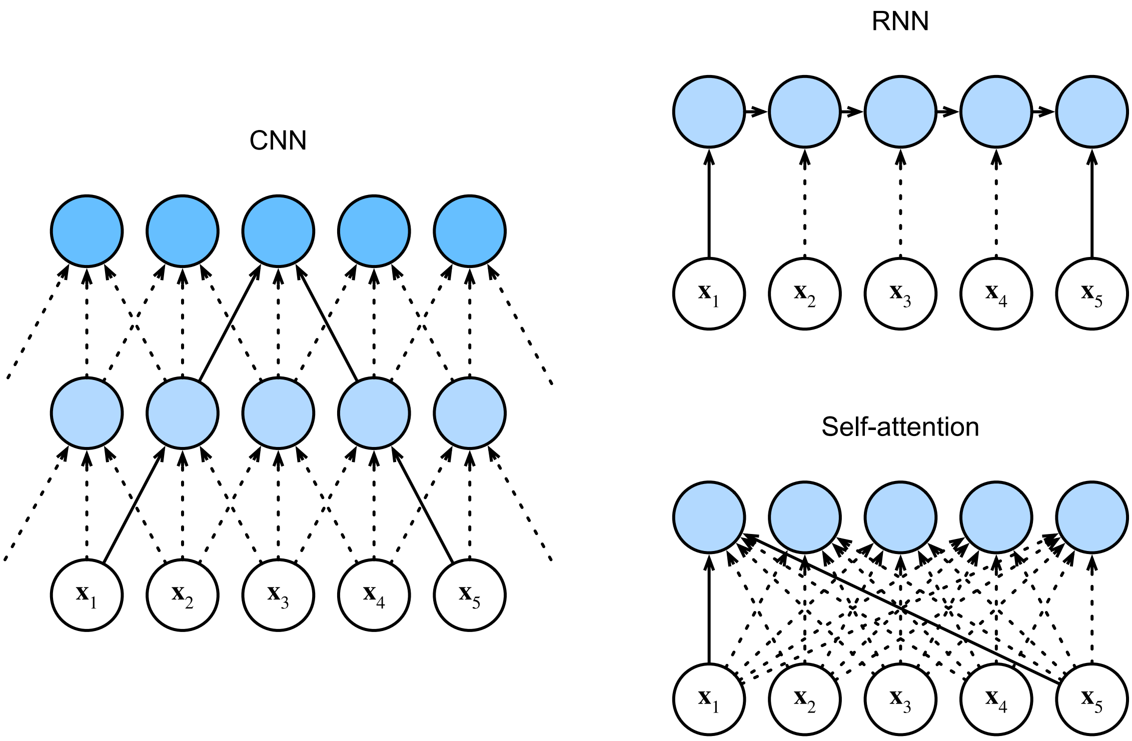


- an architecture commonly used in NLP and other types of tasks
- Encoder: take raw input and represent the input as tensors after processing(could be word2vec, neural layers, attention...)
- Decoder: mainly for outputting the result to desire form([0, 1], probability distribution, classification, etc)

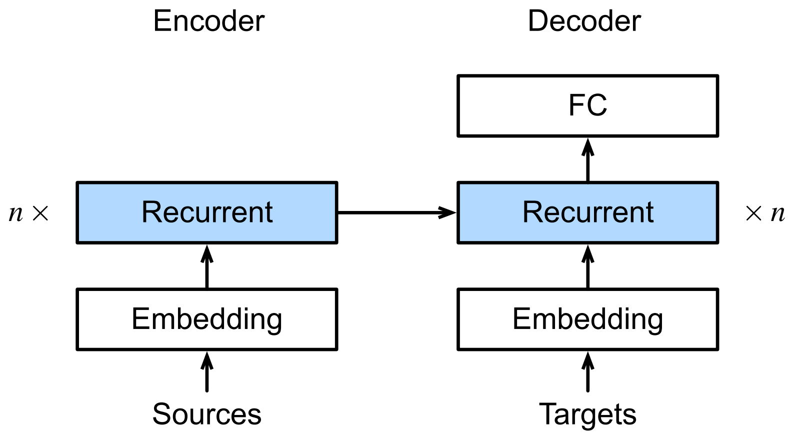
```python
from torch import nn

class Encoder(nn.Module):
    def __init__(self, **kwargs):
        super(Encoder, self).__init__(**kwargs)

    def forward(self, X, *args):
        raise NotImplementedError

class Decoder(nn.Module):
    def __init__(self, **kwargs):
        super(Decoder, self).__init__(**kwargs)

    def init_state(self, encoder_outputs, *args):
        raise NotImplementedError

    def forward(self, X, state):
        raise NotImplementedError
```

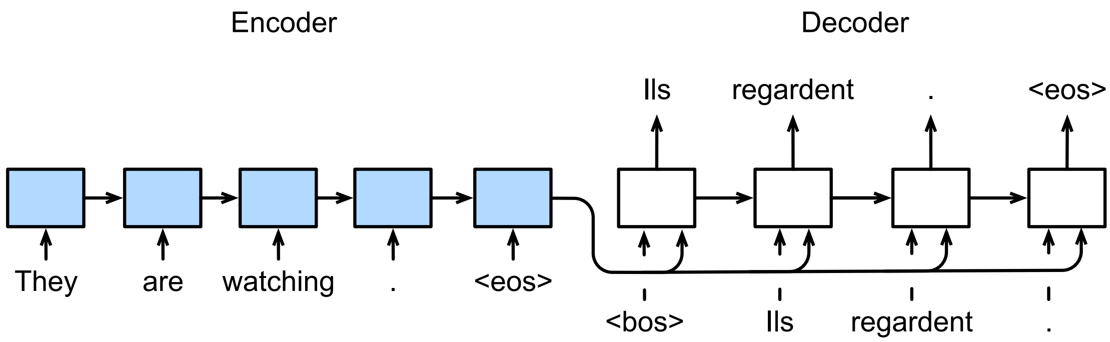
Seq2Seq Learning

- A specific type of tasks whose input and output are both sequences of any length
- Ex. Machine Translation
- Common arch of seq2seq models:



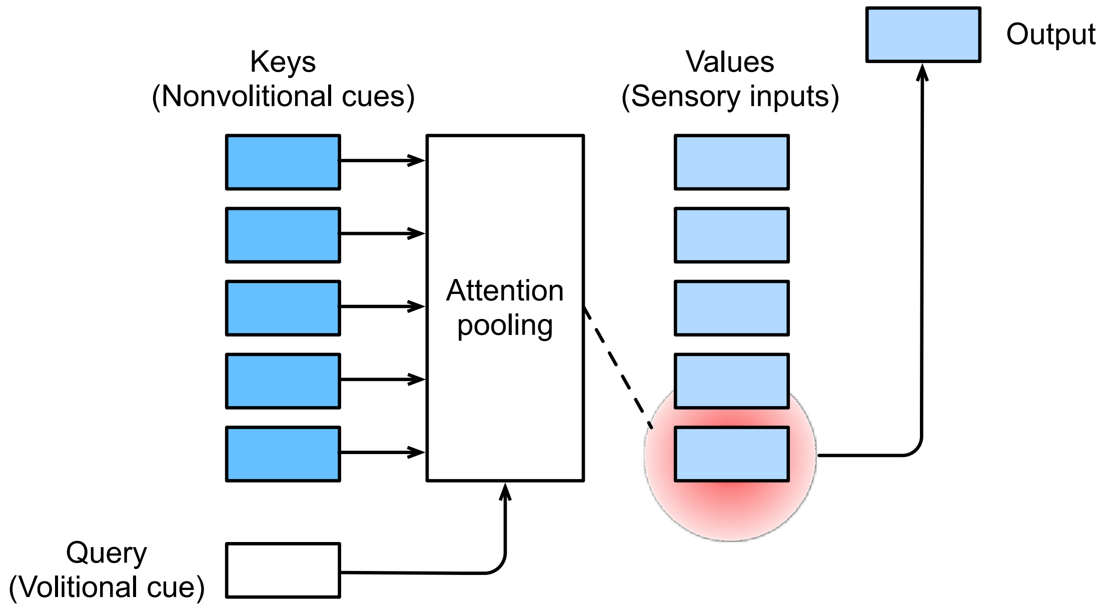
- Machine Translation using RNN





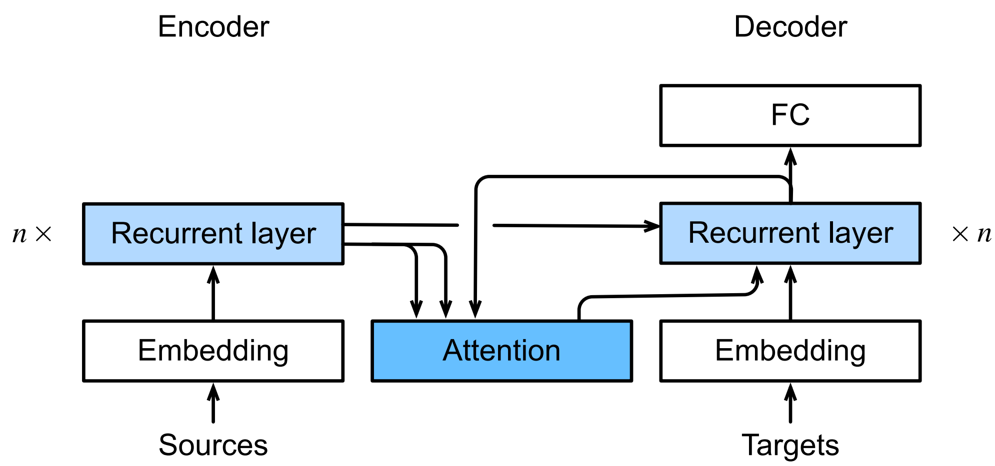
- BLEU(Bilingual Evaluation Understudy) for machine translation
  - formula: $\exp\left(\min\left(0, 1 - \frac{\mathrm{len}_{\mathrm{label}}}{\mathrm{len}_{\mathrm{pred}}}\right)\right) \prod_{n=1}^{k} p_n^{1/2^n}$
  - where p_n represent the `n-gram` accuracy

Attention Mechanism & Attention Score

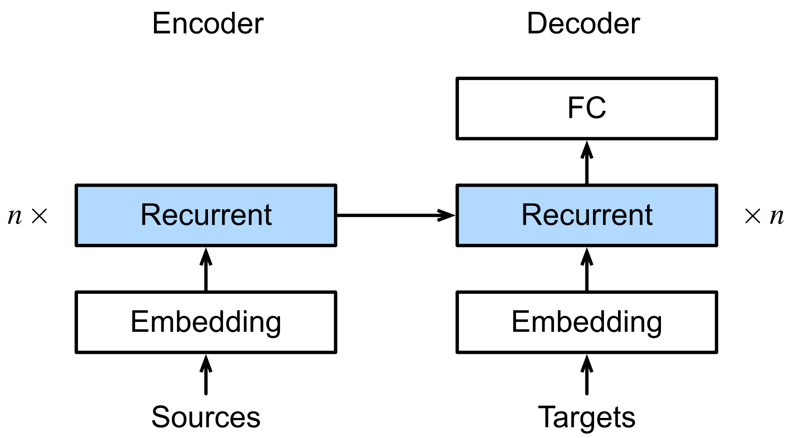


- Attention Mechanism, **KVQ**
  - `Key`: what is presented
  - `Value`: sensory inputs(?)
  - `Query`: what we are interested
  - The idea is to using Query to find "important" `Key`s
- Attention Score, $\alpha(x, x_i)$
  - model the relationship(importance, similarity) of `Keys` & `Querys`
  - Kernel Regression
    - $\alpha(x, x_i) = \frac{K(x-x_i)}{\sum_{j=1}^{n} K(x-x_j)}$
  - Additive Attention
    - $a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q\mathbf{q} + \mathbf{W}_k\mathbf{k}) \in \mathbb{R},$
  - Scaled Dot-Product Attention
    - $a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top\mathbf{k}/\sqrt{d}$
    - Matrix form: $\mathrm{softmax}\left(\frac{\mathbf{QK}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n \times v}.$

Seq2Seq with Attention



- Notice the difference with



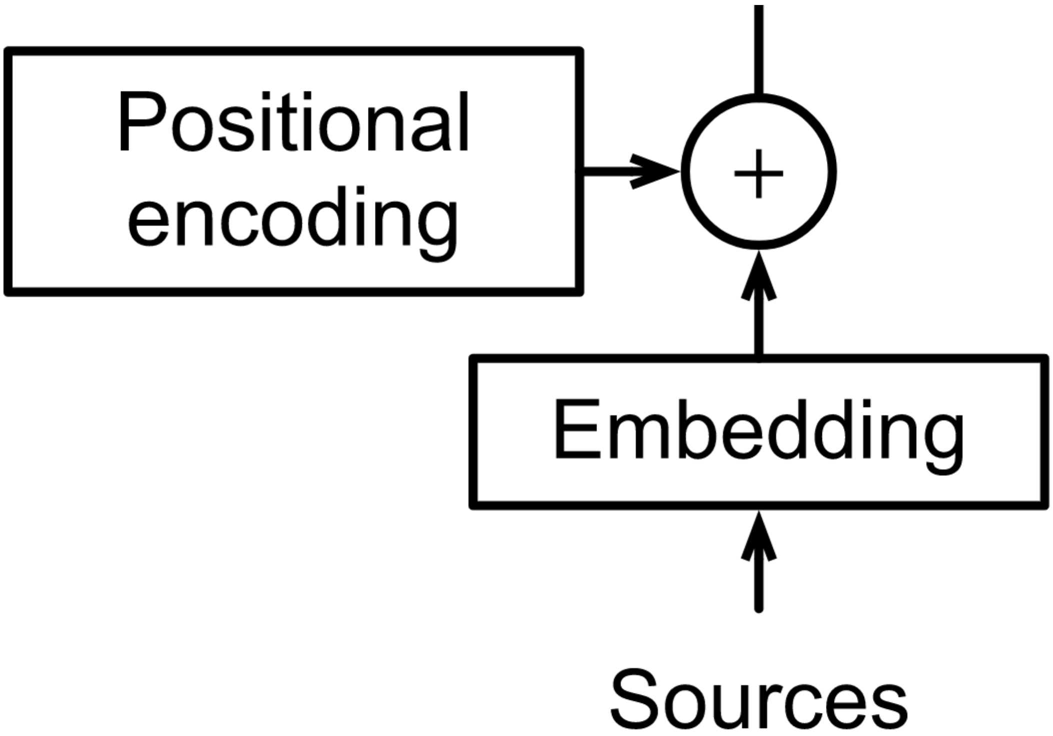
- Here, the `Query` is decoder's input, `Key` & `Value` are both encoders output (final hidden state)

Self-attention

- Self-attention means $Queries = Values = Keys = X(input)$
- So we are trying to find the relationship between one token x_i with other tokens
- $\mathbf{y}_i = f(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_n, \mathbf{x}_n)) \in \mathbb{R}^d$, where x_i is `Query` and (x_j, x_j) is `Key-Value`

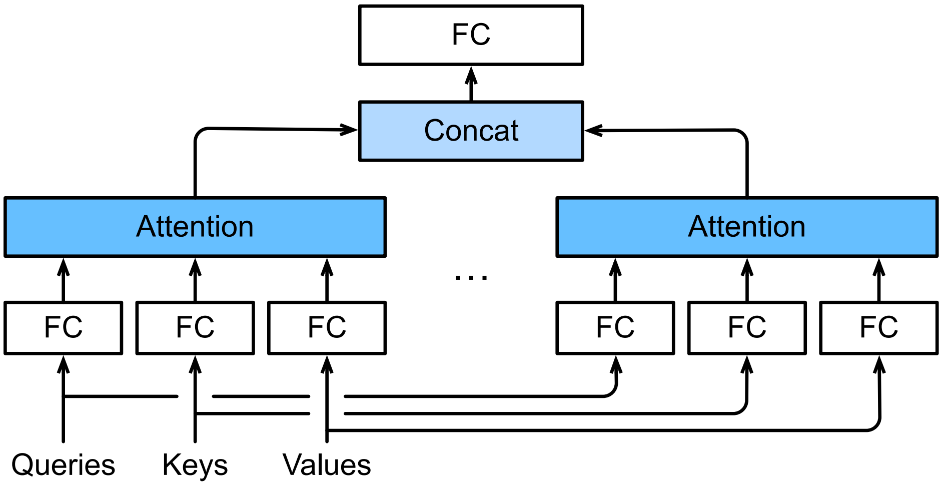
Position Encoding

- Self-attention does not contain information about relative positions (of tokens)
- Position Encoding aims to "encode" some relative position information to the input X



- A commonly used position encoding method is using these sin and cos
  - for the Position Encoding Matrix P
  - $P_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right)$
  - $P_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right)$

Multi-head Attention



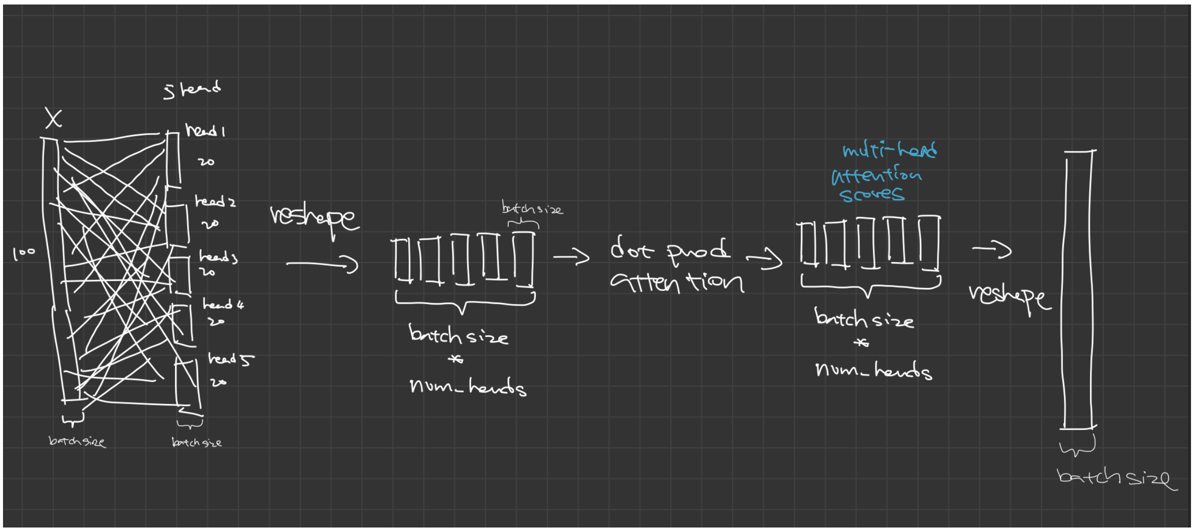
- Multi-head Attention aims to capture different "relationships" between `Query` and `Key` using multiple parallel attention layers and concat them to get the final result.
- Mathematically:
  - $\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathbb{R}^{p_v}$, where f is some kind of attention function and h_i is the i_{th} head
  - $result = \mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix} \in \mathbb{R}^{p_o}$

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 num_heads, dropout, bias=False, **kwargs):
        super(MultiHeadAttention, self).__init__(**kwargs)
        self.num_heads = num_heads
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = nn.Linear(query_size, num_hiddens, bias=bias)
        self.W_k = nn.Linear(key_size, num_hiddens, bias=bias)
        self.W_v = nn.Linear(value_size, num_hiddens, bias=bias)
        self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)

    def forward(self, queries, keys, values, valid_lens):
        # assuming num_queries = num_keys = num_values

        # initial queries:
        # (batch_size, num_queries, num_hiddens)
        # transformed queries:
        # (batch_size * num_heads, num_queries, num_hiddens/num_heads)
        queries = transpose_qkv(self.W_q(queries), self.num_heads)
        keys = transpose_qkv(self.W_k(keys), self.num_heads)
        values = transpose_qkv(self.W_v(values), self.num_heads)

        if valid_lens is not None:
            valid_lens = torch.repeat_interleave(
                valid_lens, repeats=self.num_heads, dim=0)

        # (batch_size * num_heads, num_queries, num_hiddens/num_heads)
        output = self.attention(queries, keys, values, valid_lens)

        # (batch_size, num_queries, num_hiddens)
```

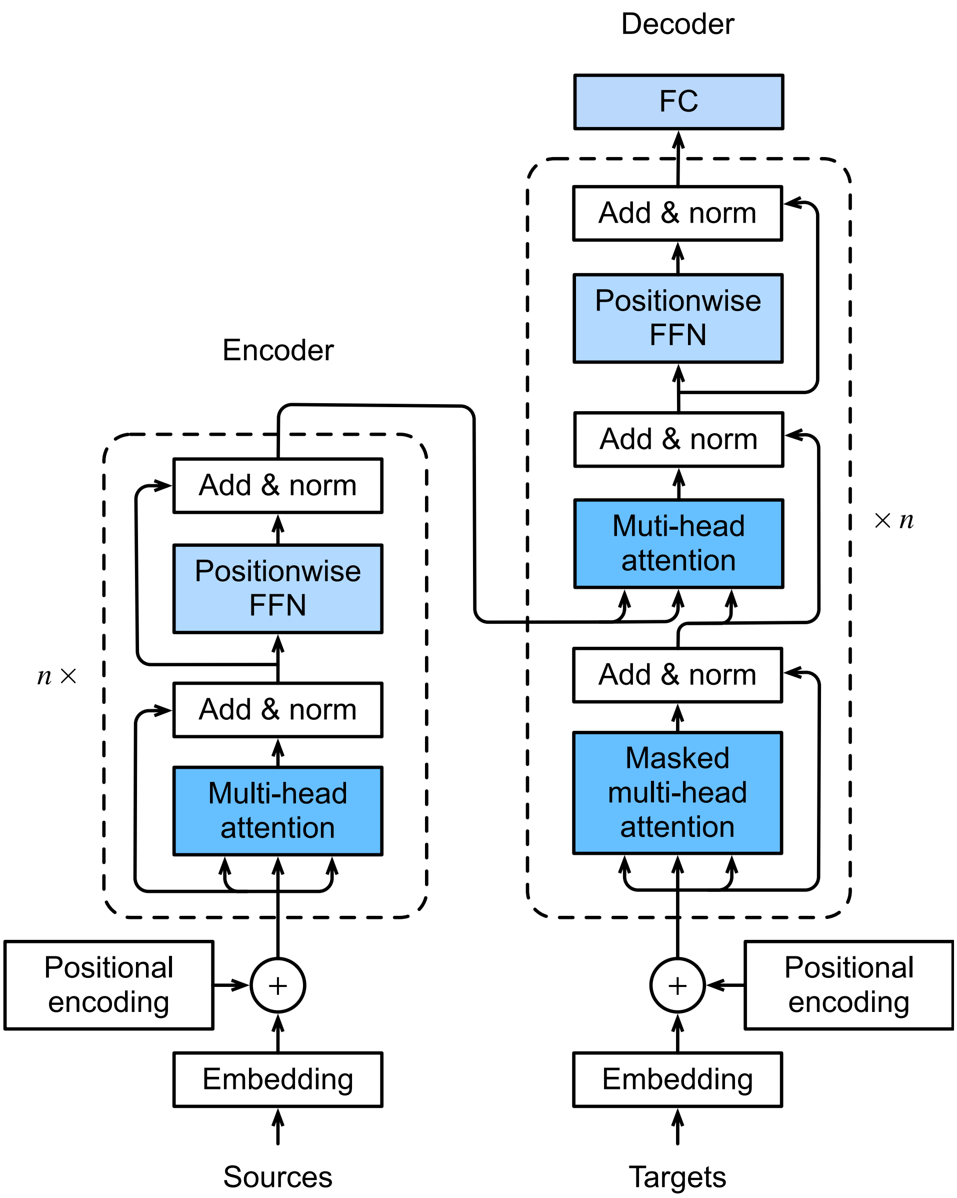
```python
        output_concat = transpose_output(output, self.num_heads)
        return self.W_o(output_concat)
```

```python
def transpose_qkv(X, num_heads):
    # (batch_size, num_queries, num_hiddens)

    # (batch_size, num_queries, num_heads, num_hiddens/num_heads)
    X = X.reshape(X.shape[0], X.shape[1], num_heads, -1)

    # (batch_size, num_heads, num_queries, num_hiddens/num_heads)
    X = X.permute(0, 2, 1, 3)

    # (batch_size * num_heads, num_queries, num_hiddens/num_heads)
    return X.reshape(-1, X.shape[2], X.shape[3])


def transpose_output(X, num_heads):
    """ reverse `transpose_qkv` """
    X = X.reshape(-1, num_heads, X.shape[1], X.shape[2])
    X = X.permute(0, 2, 1, 3)
    return X.reshape(X.shape[0], X.shape[1], -1)
```

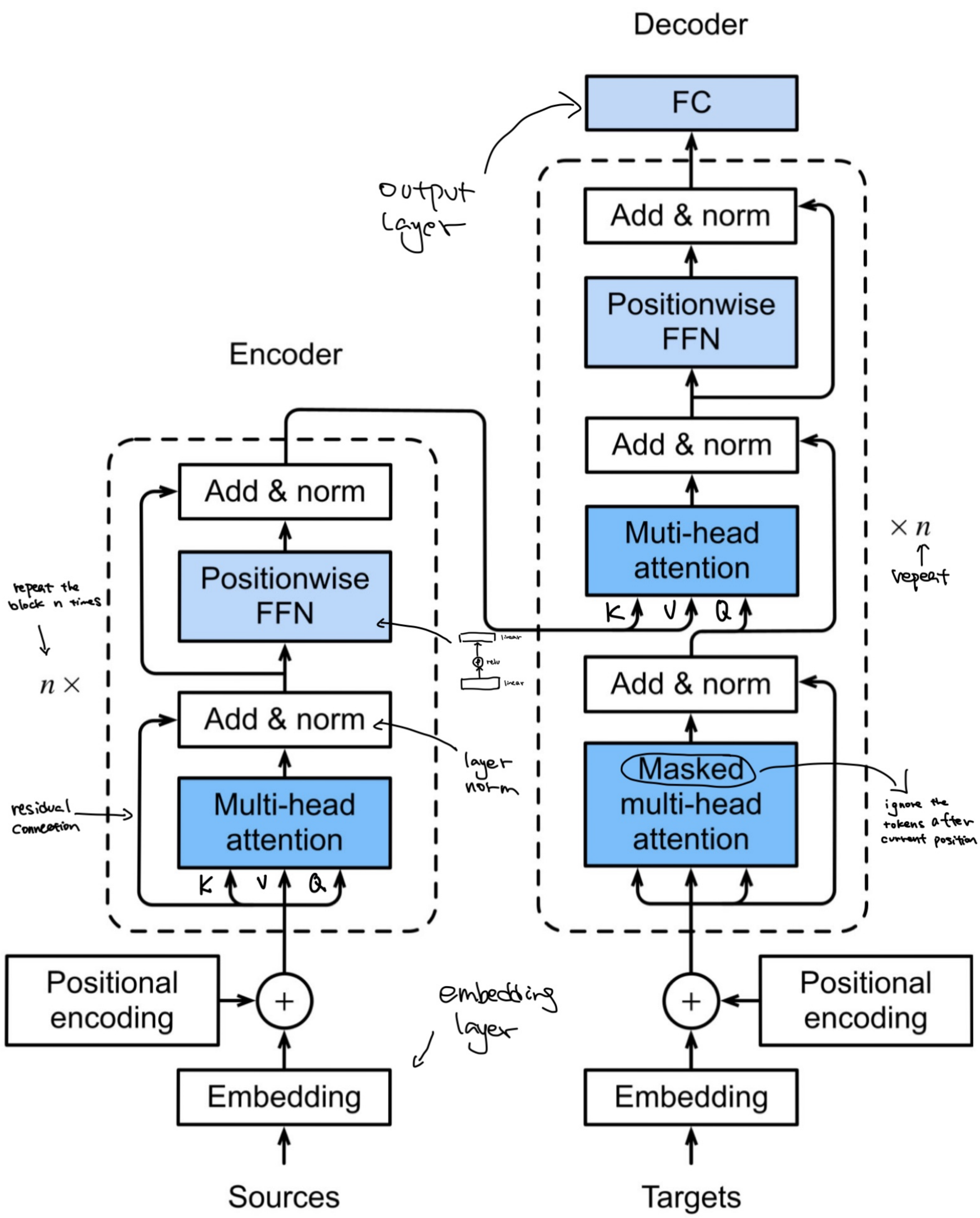
- Shaping



Transformer



Annotated graph


Decoder
FC
output layer
Add & norm
Positionwise FFN
Add & norm
Muti-head attention
K V Q
× n
repeat
Add & norm
Masked multi-head attention
ignore the tokens after current position
Positional encoding
Embedding
Targets
Encoder
Add & norm
Positionwise FFN
Add & norm
Multi-head attention
K V Q
repeat the block n times
n ×
residual connection
layer norm
Positional encoding
Embedding
embedding layer
Sources