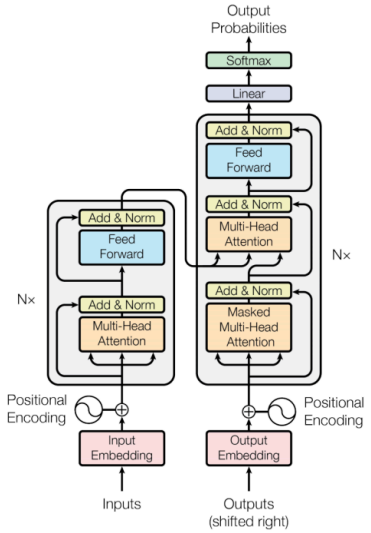
I. Tổng quan

Transformer là một kiến trúc quan trọng và phổ biến các mô hình học sâu, hoạt động theo cơ chế self attention. Kiến trúc này có thể giải quyết các vấn đề thường gặp trong RNNs và LTSMs, giúp mô hình xử lý các bộ dữ liệu tuần tự (sequential data) linh hoạt và hiệu quả hơn. Transformer có hai phần chính: Encoder và Decoder. Mỗi phần sẽ có 3 lớp cơ bản: lớp positional encoding để thêm thông tin vị trí của các từ, lớp multi head self attention dùng để ngữ cảnh hóa các tokens, lớp add and layer normalization dùng để chuẩn hóa và lớp FFN dùng để xử lý các giá trị của attention.



Ưu điểm:

- Xử lý song song: Transformer có thể xử lý đồng thời nhiều phân đoạn của một tệp dữ liệu, thay vì xử lý tuần tự như RNNs
- Long-Range Dependencies: self-attention giúp mô hình phát hiện các sự ràng buộc có khoảng cách lớn giữa các dữ liệu, khiến cho các công việc như NLP (khi một từ trong một đoạn văn bản có nhiều nghĩa và phải dựa vào ngữ cảnh của văn bản để hiểu) trở nên dễ dàng hơn.

II. Xây dựng các thành lớp

1. Lớp Positional Encoding

Transformer không sử dụng RNN hay CNN nên nó không tự biết thứ tự của từ. Nếu sử dụng embeddings, sẽ rất khó phát hiện các từ lặp lại có nghĩa khác nhau trong văn bản. Do vậy, lớp Positional Encoding có vai trò cộng thêm embedding vị trí vào ma trận ngữ nghĩa (meaning vector) để chuyển thành ma trận ngữ cảnh (context vector), có khả năng xác định được vị trí của các tokens. Công thức tính được biểu diễn như sau:

$$p_t^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k \cdot t), & \text{if } i = 2k \\ \cos(\omega_k \cdot t), & \text{if } i = 2k+1 \end{cases} \quad \text{where} \quad \omega_k = \frac{1}{10000^{2k/d}}$$

Trong đó, t là vị trí của token, i là vị trí không gian của nó trong vector . Với i là giá trị chẵn hoặc lẻ, ta có thể dùng một trong hai hàm sin hoặc cos tương ứng.

```python
class PositionalEncoding(nn.Module):  1 usage
    def __init__(self, d_model, max_len=5000):
        super(PositionalEncoding, self).__init__()
        # Tạo ma trận [max_len, d_model] chứa thông tin vị trí
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))

        # Áp dụng công thức Sinusoidal
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        # Thêm chiều batch: [1, max_len, d_model]
        pe = pe.unsqueeze(0)

        # Đăng ký buffer (không phải là tham số huấn luyện)
        self.register_buffer( name: 'pe', pe)

    def forward(self, x):
        # x: [batch_size, seq_len, d_model]
        # Cộng embedding của từ với positional encoding tương ứng
        x = x + self.pe[:, :x.size(1)]
        return x
```

2. Lớp Multi-head Attention + Scale-dot

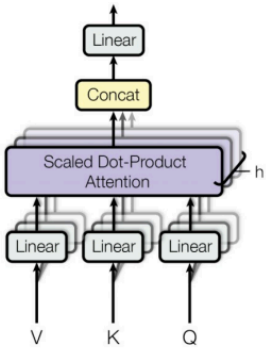
Lớp này là phần quan trọng nhất của Transformer, có nhiệm vụ tập trung vào nhiều phân đoạn của văn bản để xác định ngữ cảnh, mối quan hệ và các ràng buộc liên quan của các tokens. Multi-head attention hoạt động theo cơ chế self-attention, tức là các tokens sẽ tự kiểm tra nhau, rồi tự thay đổi ý nghĩa của mình. Từ đó, ta sẽ có một biểu diễn vector có khả năng hiểu ngữ cảnh của văn bản. Đối với lớp Decoder, ta sẽ có thêm thuộc tính Masked, dùng để “che đi” các tokens đứng đằng sau tokens hiện tại trong văn bản, giúp mô hình tạo sinh ra tokens lần lượt và chỉ sử dụng ngữ cảnh đã có ở phân đoạn trước. Multi-head Attention bao gồm nhiều head

scale-dot attention, tính toán weighted sum của các embeddings đầu vào, dựa vào giá trị Q,K,V của chúng. Công thức tính attention được biểu diễn như sau:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Với Q,K,V lần lượt là các Query, Key, Value của embeddings, được biến đổi thông qua các lớp Linear. Sau khi có giá trị của Attention được tính toán xong, ta nối các đầu lại (concat), rồi thông qua một lớp Linear cuối cùng trước khi chuyển kết quả đến lớp khác

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$




```python
class MultiHeadAttention(nn.Module):  5 usages
    def __init__(self, d_model, n_head):
        super(MultiHeadAttention, self).__init__()
        assert d_model % n_head == 0, "d_model phải chia hết cho n_head"

        self.d_head = d_model // n_head
        self.n_head = n_head
        self.d_model = d_model

        # Các lớp Linear để chiếu Q, K, V
        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)

        # Lớp Linear cuối cùng sau khi nối các head
        self.fc_out = nn.Linear(d_model, d_model)

    def forward(self, query, key, value, mask=None):
        batch_size = query.shape[0]

        # 1. Tính Q, K, V qua các lớp Linear
        Q = self.w_q(query)
        K = self.w_k(key)
        V = self.w_v(value)
```

```
# 2. Chia thành nhiều đầu (Split heads)
# Biến đổi: [batch_size, seq_len, d_model] -> [batch_size, seq_len, n_head, d_head]
# Sau đó transpose để đưa n_head lên trước: [batch_size, n_head, seq_len, d_head]
Q = Q.view(batch_size, -1, self.n_head, self.d_head).permute(0, 2, 1, 3)
K = K.view(batch_size, -1, self.n_head, self.d_head).permute(0, 2, 1, 3)
V = V.view(batch_size, -1, self.n_head, self.d_head).permute(0, 2, 1, 3)

# 3. Scaled Dot-Product Attention
# energy: [batch_size, n_head, seq_len_q, seq_len_k]
energy = torch.matmul(Q, K.permute(0, 1, 3, 2)) / math.sqrt(self.d_head)

# Áp dụng Mask (nếu có) - Dùng cho Decoder (Look-ahead mask) hoặc Padding mask
if mask is not None:
    energy = energy.masked_fill(mask == 0, -1e9)

attention = torch.softmax(energy, dim=-1)

# x: [batch_size, n_head, seq_len_q, d_head]
x = torch.matmul(attention, V)

# 4. Nối các đầu lại (Concat)
# [batch_size, seq_len_q, n_head * d_head] = [batch_size, seq_len_q, d_model]
x = x.permute(0, 2, 1, 3).contiguous().view(batch_size, -1, self.d_model)

return self.fc_out(x)
```

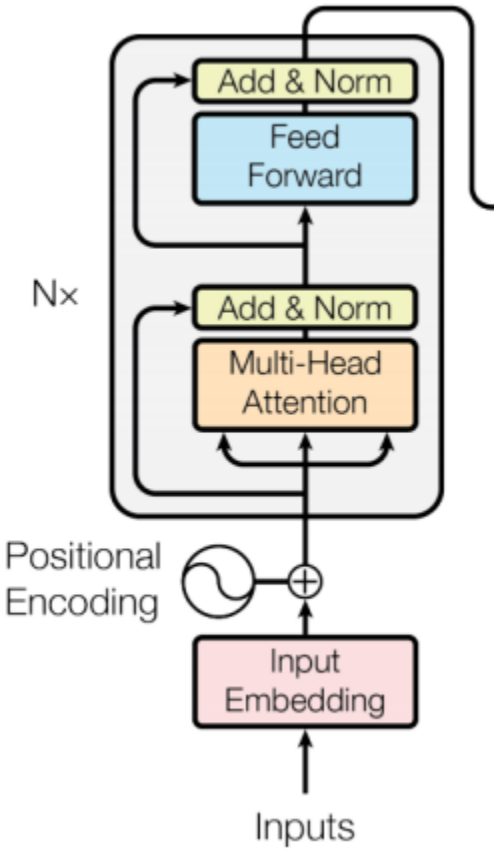
3. Lớp Positionwide Feed Forward (FFN)

Lớp này trong kiến trúc transformer là một lớp FFN đơn giản, có nhiệm vụ nhận các dữ liệu từ lớp attention, rồi chuyển hóa chúng thành định dạng mới, dùng làm đầu vào của decoder hoặc đầu ra của mô hình. (Linear -> ReLU -> Linear)

```
class PositionwiseFeedForward(nn.Module):  3 usages
    def __init__(self, d_model, d_ff):
        super(PositionwiseFeedForward, self).__init__()
        self.fc1 = nn.Linear(d_model, d_ff)
        self.fc2 = nn.Linear(d_ff, d_model)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.fc2(self.relu(self.fc1(x)))
```

4. Block Encoder hoàn chỉnh

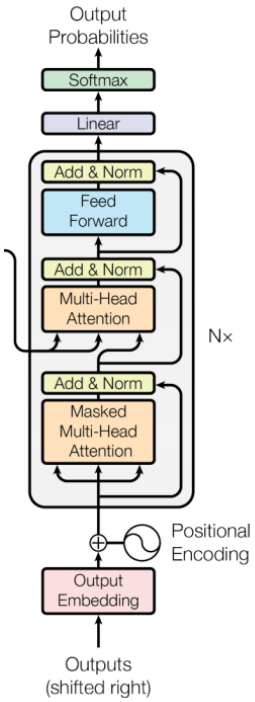
Như vậy, sau khi ghép các components ở trên lại với nhau, ta sẽ có một block encoder theo cấu trúc như hình vẽ. Embeddings có Positional Encoding, Attention và FFN với mỗi lớp sẽ đi qua một layer normalization để chuẩn hóa trước khi chuyển sang layer khác



```python
class EncoderLayer(nn.Module):  3 usages
    def __init__(self, d_model, n_head, d_ff, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, n_head)
        self.ffn = PositionwiseFeedForward(d_model, d_ff)

        # Layer Normalization
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

        self.dropout = nn.Dropout(dropout)

    def forward(self, src, src_mask):
        # Sublayer 1: Multi-Head Self-Attention
        _src = self.self_attn(src, src, src, src_mask)
        # Add & Norm
        src = self.norm1(src + self.dropout(_src))

        # Sublayer 2: Feed Forward
        _src = self.ffn(src)
        # Add & Norm
        src = self.norm2(src + self.dropout(_src))

        return src
```

5. Block Decoder hoàn chỉnh

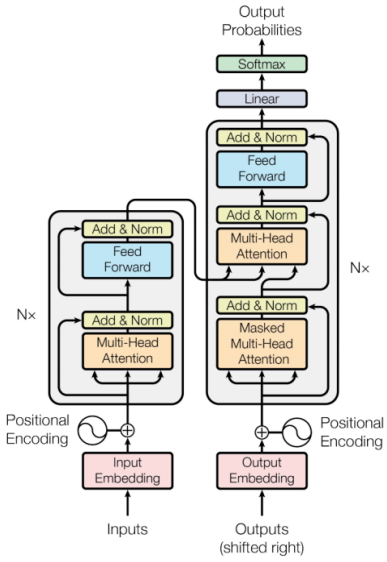
Tương tự như encoder, nhưng decoder có thêm một layer dành cho masked attention, layer attention thứ hai của decoder sẽ nhận input của encoder



```python
class DecoderLayer(nn.Module):  2 usages
    def __init__(self, d_model, n_head, d_ff, dropout):
        super(DecoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, n_head)
        self.cross_attn = MultiHeadAttention(d_model, n_head)
        self.ffn = PositionwiseFeedForward(d_model, d_ff)

        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, trg, enc_src, trg_mask, src_mask):
        # 1. Masked Multi-Head Self-Attention
        # trg_mask ở đây đảm bảo vị trí t không nhìn thấy t+1
        _trg = self.self_attn(trg, trg, trg, trg_mask)
        trg = self.norm1(trg + self.dropout(_trg))

        # 2. Multi-Head Cross-Attention
        # Query lấy từ Decoder (trg), Key & Value lấy từ Encoder (enc_src)
        _trg = self.cross_attn(trg, enc_src, enc_src, src_mask)
        trg = self.norm2(trg + self.dropout(_trg))

        # 3. Feed Forward
        _trg = self.ffn(trg)
        trg = self.norm3(trg + self.dropout(_trg))

        return trg
```

6. Transformer hoàn chỉnh

Như vậy, sau khi ghép 2 block decoder và encoder lại với nhau, ta sẽ có một transformer hoàn chỉnh:



```python
class Transformer(nn.Module):  1 usage                                                    ⚠ 2
    def __init__(self, src_vocab_size, trg_vocab_size,
                 d_model=512, n_head=8, n_layer=6, d_ff=2048,
                 dropout=0.1, max_len=100):
        super(Transformer, self).__init__()

        self.encoder = Encoder(src_vocab_size, d_model, n_layer, n_head, d_ff, dropout, max_len)
        self.decoder = Decoder(trg_vocab_size, d_model, n_layer, n_head, d_ff, dropout, max_len)

    def make_src_mask(self, src, src_pad_idx):  1 usage
        # Mask các vị trí padding trong source
        # src: [batch_size, seq_len] -> [batch_size, 1, 1, seq_len]
        return (src != src_pad_idx).unsqueeze(1).unsqueeze(2)

    def make_trg_mask(self, trg, trg_pad_idx):  1 usage
        # Mask padding trong target
        trg_pad_mask = (trg != trg_pad_idx).unsqueeze(1).unsqueeze(2)

        # Mask look-ahead (tam giác trên)
        trg_len = trg.shape[1]
        trg_sub_mask = torch.tril(torch.ones((trg_len, trg_len), device=trg.device)).bool()

        return trg_pad_mask & trg_sub_mask

    def forward(self, src, trg, src_pad_idx=0, trg_pad_idx=0):
        src_mask = self.make_src_mask(src, src_pad_idx)
        trg_mask = self.make_trg_mask(trg, trg_pad_idx)

        enc_src = self.encoder(src, src_mask)
        output = self.decoder(trg, enc_src, trg_mask, src_mask)

        return output
```