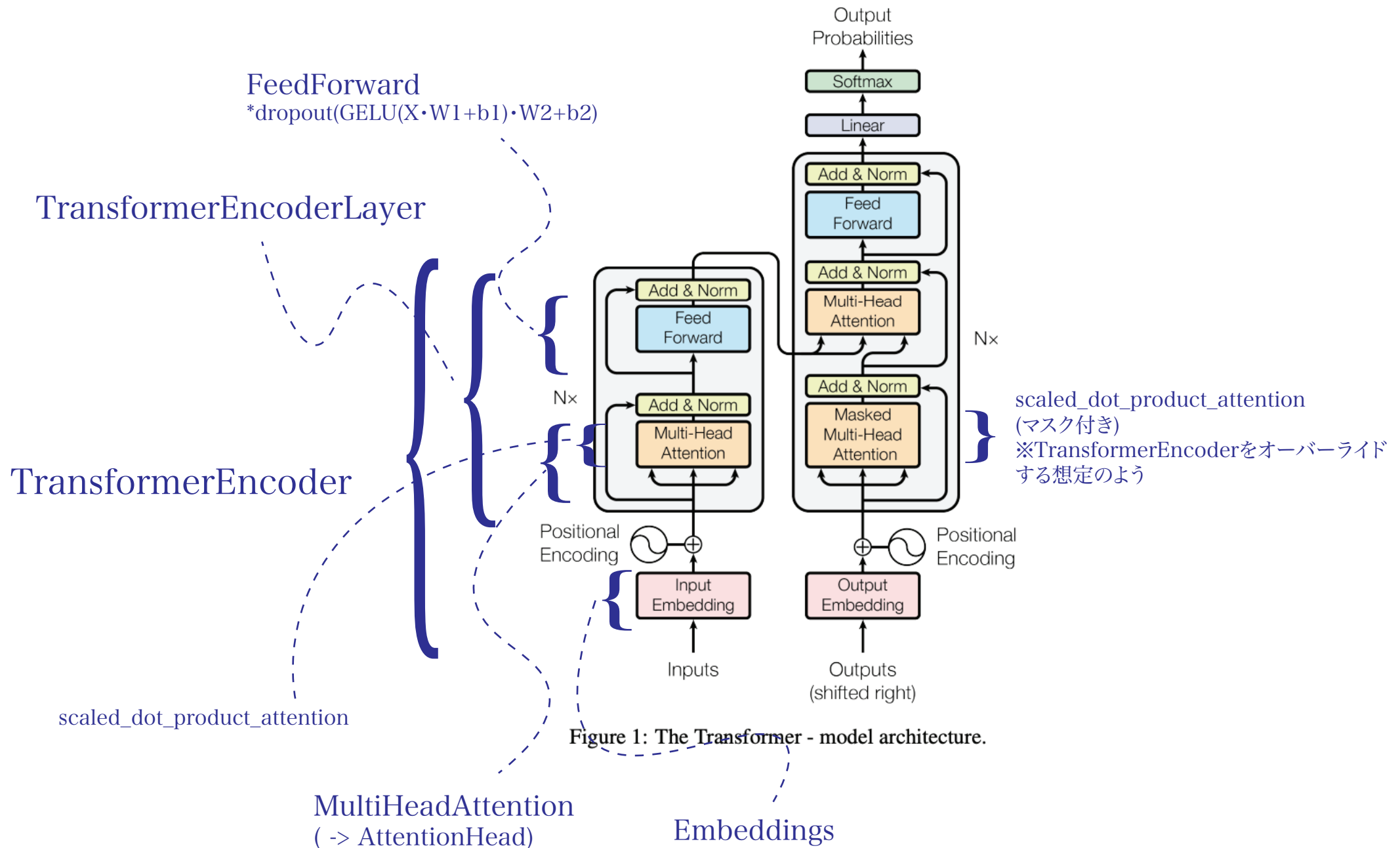


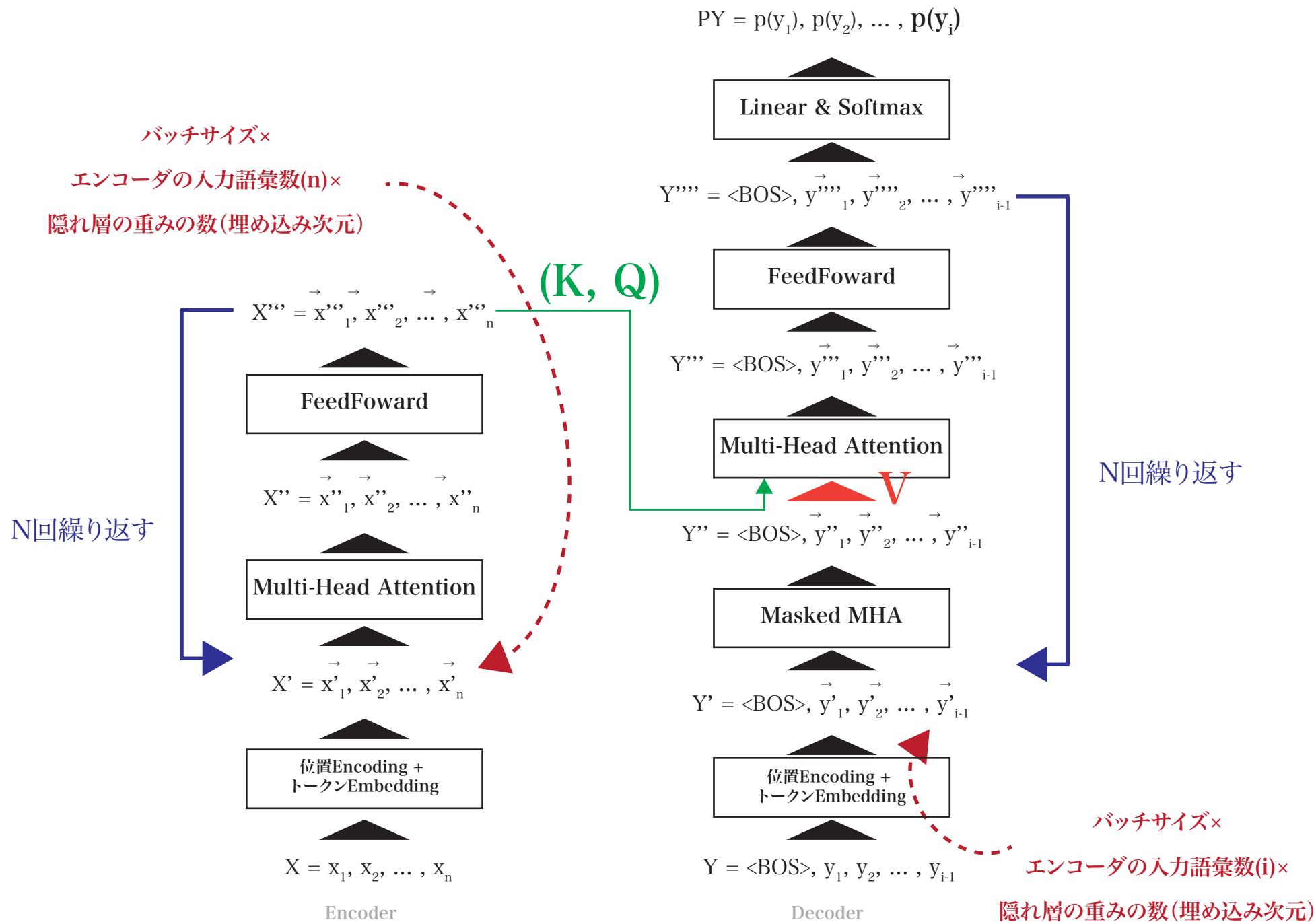
# Natural Language Processing with Transformers

## (3章 Transformerの詳細)

# “Attention Is All You Need” の図表と書籍中のコードの対応



# Transformerの大枠 (Residual Connectionと加算・正規化等は省略)



# Transformerを構成する主要なモジュール

- ①Embedding
- ②Positional Encoding
- ③Multi-Head Attention
- ④FeedForward 今日はここまで
- ⑤Masked Multi-Head Attention (Masked MHA)
- ⑥分類器

# ①Embedding／②Positional Encoding(1)

各トークンまたは位置の情報である $X = x_1, x_2, \dots, x_n$ を $X = \vec{x}_1, \vec{x}_2, \dots, \vec{x}_n$ のように変換する処理を行う。

各要素をOne-hotベクトル形式ではなく、より圧縮されたベクトルに変換し、情報を保存する。共起を想定した情報の保存方法であり、**1要素が複数の情報(トークン・位置など)を、さらに複数要素が1つの情報を表すことに寄与**する。

①トークン…単語、あるいは語幹など。トークナイザの実装次第で単位が変わる。

書籍中の文脈では、DistilBertTokenizerで抽出されたトークンということになる( $X = \text{input\_ids}$ )。

`vocab_size`:トークンのバリエーション数(30522)

```
self.token_embeddings = nn.Embedding(config.vocab_size, config.hidden_size)
```

`hidden_size`:埋め込み後の各ベクトルの長さ

```
token_embeddings = self.token_embeddings(input_ids)
```

②位置…文中において各トークンの出現する位置に関する情報。

```
self.position_embeddings = nn.Embedding(config.max_position_embeddings, config.hidden_size)
```

`max_position_embeddings`:処理可能な文章の最大長(512)

```
position_ids = torch.arange(seq_length, dtype=torch.long).unsqueeze(0)
```

`seq_length`:入力された文章の長さ

```
position_embeddings = self.position_embeddings(position_ids)
```

`position_ids`:入力された文章に含まれるトークンを頭から0,1,2,...と数え上げた数字の配列

# ①Embedding／②Positional Encoding(2)

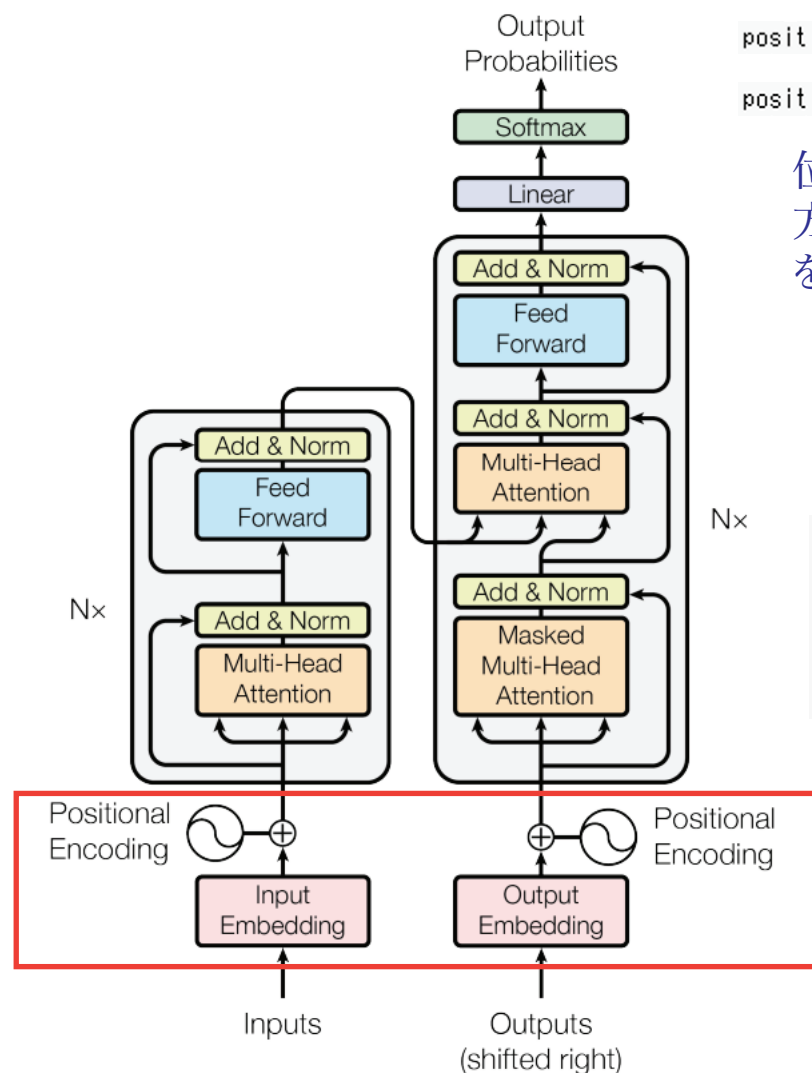


Figure 1: The Transformer - model architecture.

```
self.position_embeddings = nn.Embedding(config.max_position_embeddings, config.hidden_size)
position_ids = torch.arange(seq_length, dtype=torch.long).unsqueeze(0)
position_embeddings = self.position_embeddings(position_ids)
```

位置埋め込みの方法はどうやら複数通り考えられるらしく……論文の方では以下のように奇数番目/偶数番目をsin/cosで分かつ埋め込み方をしている。任意長の文章に対応できることがメリットらしい？

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

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

```
embeddings = token_embeddings + position_embeddings
embeddings = self.layer_norm(embeddings)
embeddings = self.dropout(embeddings)
return embeddings
```

## ▼ 悩みポイント

トークン埋め込みと位置埋め込み同士の足し算で良い理由がよくわからない。

トークンと位置の埋め込み情報を混ぜるのは直感的には望ましくないようにも見えるが、、

### ③Multi-Head Attention (MHA)(1)

Multi-Head AttentionはAttention-Head、もとい**Scaled Dot Product Attention**をh個束ねた層である。

(書籍/BERT中の $h = \text{num\_heads} = 12$ 、論文  
中では $h = 8$ )

$\text{MHA}(Q, K, V) \rightarrow \text{concat}(\text{head}_1, \text{head}_2, \dots, \text{head}_h) \cdot W^o$

$\text{head}_i = \text{Attention}(Q \cdot W_i^Q, K \cdot W_i^K, V \cdot W_i^V)$

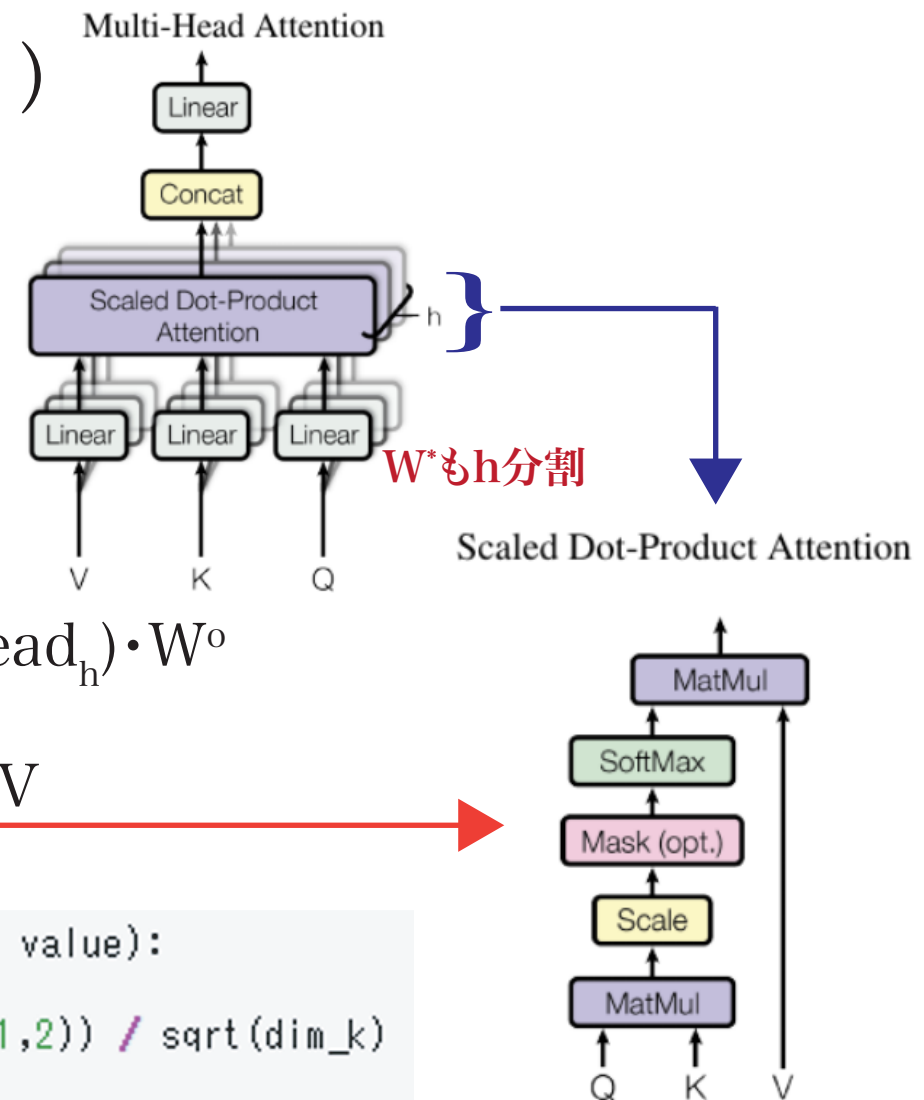
$\text{Attention}(Q, K, V) = \text{softmax}(Q \cdot K^T / \sqrt{d}) \cdot V$

**scaled\_dot\_product\_attention**

```
def scaled_dot_product_attention(query, key, value):  
    dim_k = query.size(-1)  
    scores = torch.bmm(query, key.transpose(1,2)) / sqrt(dim_k)  
    weights = F.softmax(scores, dim=-1)  
    return torch.bmm(weights, value)
```

```
class AttentionHead(nn.Module):  
    def __init__(self, embed_dim, head_dim):  
        super().__init__()  
        self.q = nn.Linear(embed_dim, head_dim)  
        self.k = nn.Linear(embed_dim, head_dim)  
        self.v = nn.Linear(embed_dim, head_dim)  
  
    def forward(self, hidden_state):  
        attn_outputs = scaled_dot_product_attention(self.q(hidden_state), self.k(hidden_state), self.v(hidden_state))  
        return attn_outputs
```

Q, K, Vそれぞれに線形変換を施し、Scaled Dot Product Attentionを計算している。





### ③Multi-Head Attention (MHA)(2)

各Attention Headの処理は(埋め込みベクトル長×埋め込みベクトル長/h)の行列を吐く。**QとKの類似度の割合に基づくVの重み付き和**。

```
class AttentionHead(nn.Module):
    def __init__(self, embed_dim, head_dim):
        super().__init__()
        self.q = nn.Linear(embed_dim, head_dim)
        self.k = nn.Linear(embed_dim, head_dim)
        self.v = nn.Linear(embed_dim, head_dim)

    def forward(self, hidden_state):
        attn_outputs = scaled_dot_product_attention(self.q(hidden_state), self.k(hidden_state), self.v(hidden_state))
        return attn_outputs
```

※列の長さはTransformerの実装次第で変化する。  
論文中でも $d_{\text{model}} \times h_d$ としている。  
なお $d_v$ は各AttentionHeadの出力列の長さ。

一方、Multi-Head AttentionはAttention Head×hをModuleListで束ねて保持し、処理の結果として(埋め込みベクトル長×埋め込みベクトル長)の行列を吐く。

```
embed_dim = config.hidden_size
num_heads = config.num_attention_heads
head_dim = embed_dim // num_heads
self.heads = nn.ModuleList([AttentionHead(embed_dim, head_dim) for _ in range(num_heads)])
self.output_linear = nn.Linear(embed_dim, embed_dim)

x = torch.cat([h(hidden_state) for h in self.heads], dim=-1)
x = self.output_linear(x)
return x
```



## ④FeedForward

全結合の2層からなる。

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

論文の中では一層目の出力にReLUを適用しているが、書籍中ではGELUを適用している。

```
class FeedForward(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.linear_1 = nn.Linear(config.hidden_size, config.intermediate_size)
        self.linear_2 = nn.Linear(config.intermediate_size, config.hidden_size)
        self.gelu = nn.GELU()
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, x):
        x = self.linear_1(x)
        x = self.gelu(x)
        x = self.linear_2(x)
        x = self.dropout(x)
        return x
```

intermediate\_size:中間層の結合数(512\*6 = 3072 ??)  
※論文の中では2048の模様

## ⑤Masked Multi-Head Attention

aaa

## ⑥分類器

aaa

まとめ

aaa

# 個人的な積み残しメモ

- ・3章の実装コードが断片的だったので、一気通貫での実装イメージがまだついていない。この悩みは10章で解決するかも？

- ・学習方法がよくわからない(習熟者には自明なのかもしれないが、、、)。

(・あとはスライド中に疑問点のメモあり)