HW11 Transformer

M113040105 劉東霖

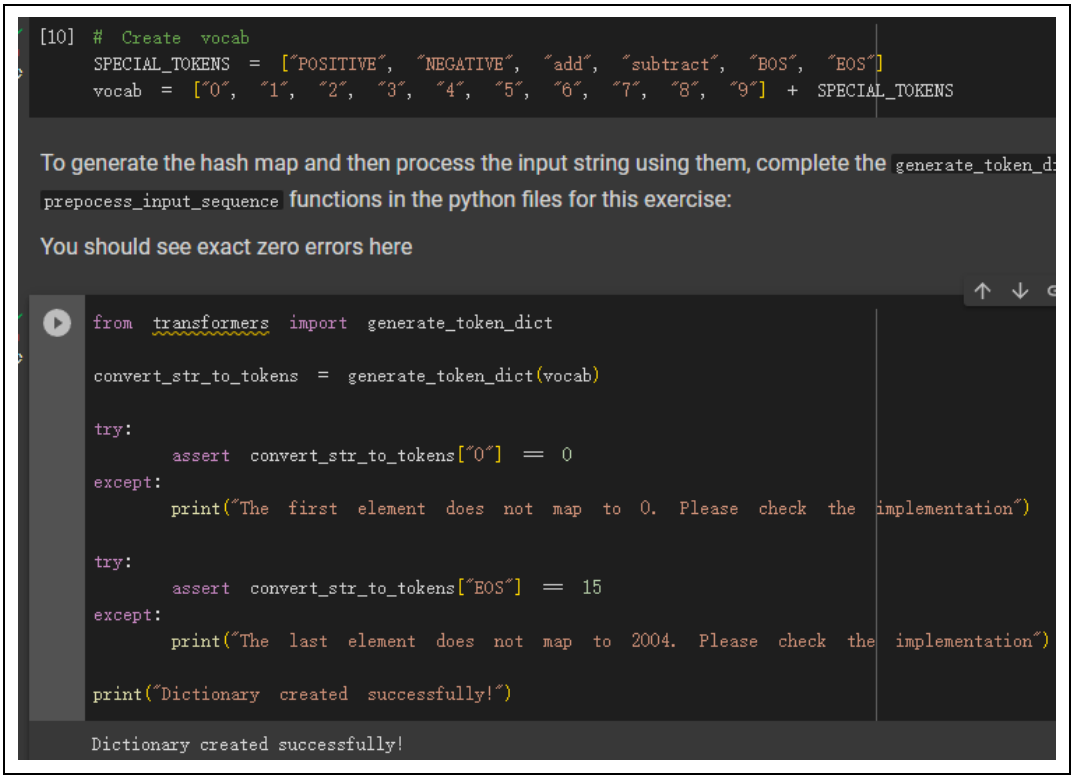
Part I. Preparation

一. generate_token_dict:

主要的目的是 string 給予獨一的編號。例如:{ ‘postive’ :0 ,’ BOS’ :1 ,’ EOS’ :2 ,……}。程式碼如下:

```
token_dict={vocab[i]:i  for  i  in  range(0,len(vocab))}
```

執行結果如下，vocob 對應到的編號都一樣。



二. prepocess_input_sequence:

這裡主要的目的是要把每個字符給予編號。

1. 首先，先把 input_str 用 split 把空白去掉
2. 如果元素是數字，則將數字拆分成單個字符，並將這些字符以 list 的形式加到 temp 中。例如，如果 i 是"123"，則會將"1"、"2"和"3"這三個字符分別添加到 temp 中。如果元素不是數字，則直接將該元素添加到 temp 中。
3. 使用在 generate_token_dict 輸出的 token_dict，將每一個字符給予相對應的編號。

程式碼如下:

```
split=input_str.split()
temp=[]
for  i  in  split:
    if  i.isnumeric():
        temp+=(list(i))
    else:
        temp.append(i)
for  i  in  temp:
    out.append(token_dict[i])
```

執行結果如下:

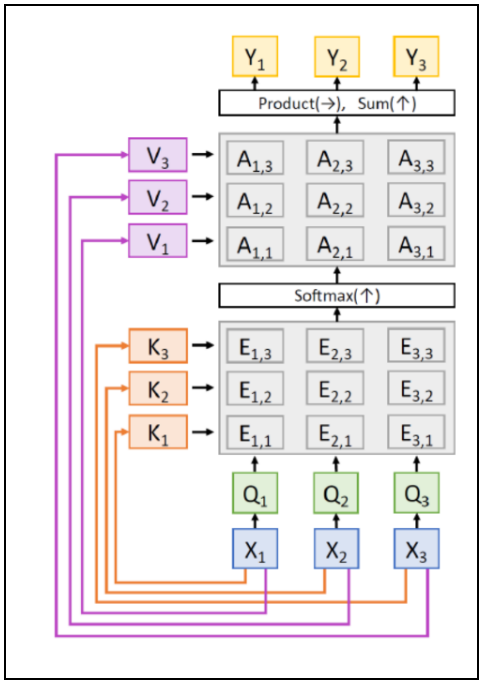
```
preprocess input token error 1:  0.0
preprocess input token error 2:  0.0
preprocess input token error 3:  0.0
preprocess input token error 4:  0.0


preprocess output token error 1:  0.0
preprocess output token error 2:  0.0
preprocess output token error 3:  0.0
preprocess output token error 4:  0.0
```

Part II. Implementing Transformer building blocks

一. scaled_dot_product_two_loop_single:

這裡主要是用雙迴圈的方式完成單一個 self attention block 計算。Self attention 的計算方式如下圖所示，Q 和 K 和 V 都是題目給定:



1. 獲取 Q 的形狀，並將其分別賦值給 k 和 m 變數。

2. 創建一個形狀為(k, k)的全零 qk 張量。
3. 用一個雙層迴圈將 Q 和 K 內積，並把結果存到 qk。
4. 將 qk 除以$\sqrt{M}$，為了防止數值過大。
5. 對 qk 的每一個 column 進行 softmax 運算。
6. 將 qk 跟 V 矩陣相乘就是輸出結果。

程式如下:

```
k,m=query.shape
qk=torch.zeros(size=(k,k))
for  i  in  range(k):
    for  j  in  range(k):
        qk[i,j]=torch.inner(query[i],key[j])
qk=qk/m**0.5
qk=torch.softmax(qk,dim=1)
out=qk  @  value
```

執行結果如下:

```
N = 2   # Number of sentences
K = 5   # Number of words in a sentence
M = 4   # feature dimension of each word embedding

query = torch.linspace(-0.4, 0.6, steps=K * M).reshape(K, M)    # **to_double_cuda
key = torch.linspace(-0.8, 0.5, steps=K * M).reshape(K, M)    # **to_double_cuda
value = torch.linspace(-0.3, 0.8, steps=K * M).reshape(K, M)    # *to_double_cuda

y = scaled_dot_product_two_loop_single(query, key, value)
y_expected = torch.tensor(
        [
                [0.08283, 0.14073, 0.19862, 0.25652],
                [0.13518, 0.19308, 0.25097, 0.30887],
                [0.18848, 0.24637, 0.30427, 0.36216],
                [0.24091, 0.29881, 0.35670, 0.41460],
                [0.29081, 0.34871, 0.40660, 0.46450],
        ]
).to(torch.float32)
print("sacled_dot_product_two_loop_single error: ", rel_error(y_expected, y))
```

```
sacled_dot_product_two_loop_single error:  5.204997002435008e-06
```

二. scaled_dot_product_two_loop_batch:

這裡是上一個的延伸，考慮到有多個輸入，流程跟上一個一樣。

1. 創建一個形狀為(n, k, k)的全零 qk 張量。
2. 創立一個雙層迴圈，利用 torch.einsum 的方式將第 i 個 query 和第 j 個 key 矩陣相乘，並將結果存到 qk。
3. 將 qk 除以$\sqrt{M}$，為了防止數值過大。
4. 對 qk 的每一個 column 進行 softmax 運算。
5. 將 qk 跟 V 矩陣相乘就是輸出結果。

程式如下:

```
qk=torch.zeros(size=(N,K,K))
for  i  in  range(K):
    for  j  in  range(K):
        qk[:,  i,  j]  =  torch.einsum('bi,bi->b',  query[:,  i],  key[:,  j])
qk=qk/M**0.5
qk=torch.softmax(qk,dim=-1)
out=torch.bmm(qk,value)
```

執行結果如下:

```
y = scaled_dot_product_two_loop_batch(query, key, value)
y_expected = torch.tensor(
        [
                [
                        [-0.09603, -0.06782, -0.03962, -0.01141],
                        [-0.08991, -0.06170, -0.03350, -0.00529],
                        [-0.08376, -0.05556, -0.02735, 0.00085],
                        [-0.07760, -0.04939, -0.02119, 0.00702],
                        [-0.07143, -0.04322, -0.01502, 0.01319],
                ],
                [
                        [0.49884, 0.52705, 0.55525, 0.58346],
                        [0.50499, 0.53319, 0.56140, 0.58960],
                        [0.51111, 0.53931, 0.56752, 0.59572],
                        [0.51718, 0.54539, 0.57359, 0.60180],
                        [0.52321, 0.55141, 0.57962, 0.60782],
                ],
        ]
).to(torch.float32)
print("scaled_dot_product_two_loop_batch error: ", rel_error(y_expected, y))
```

```
scaled_dot_product_two_loop_batch error: 4.020571992067902e-06
```

三. scaled_dot_product_no_loop_batch:

延續上一個內容，只是這裡引入 mask。Mask 用於遮罩掉未來位置的信息，以確保模型只能關注當前或過去的元素。這樣模型在生成序列的過程中只能依賴於當前或過去的信息，不會受到未來信息的影響。

1. 先用 self attention 的公式算出 qk。
2. 假如有遮罩的話，就將 qk 在 mask 為 True 的地方填滿10^{-9}，這樣在後面進行指數運算時就不會參考到訊息了。
3. 對 qk 的每一個 column 進行 softmax 運算。
4. 將 qk 跟 V 矩陣相乘就是輸出結果。

程式碼如下:

```
qk=torch.bmm(query, key.transpose(1, 2))
qk=qk/M**0.5
if mask is not None:
        ################################################################################
        # TODO: Apply the mask to the weight matrix by assigning -1e9 to the
        # positions where the mask value is True, otherwise keep it as it is.
        ################################################################################
        # Replace "pass" statement with your code
    qk = qk.masked_fill(mask, -1e9)

weights_softmax = torch.softmax(qk, dim=-1)
y = torch.bmm(weights_softmax, value)
```

執行結果如下:

```
y_expected = torch.tensor(
        [
                [
                        [-0.09603, -0.06782, -0.03962, -0.01141],
                        [-0.08991, -0.06170, -0.03350, -0.00529],
                        [-0.08376, -0.05556, -0.02735, 0.00085],
                        [-0.07760, -0.04939, -0.02119, 0.00702],
                        [-0.07143, -0.04322, -0.01502, 0.01319],
                ],
                [
                        [0.49884, 0.52705, 0.55525, 0.58346],
                        [0.50499, 0.53319, 0.56140, 0.58960],
                        [0.51111, 0.53931, 0.56752, 0.59572],
                        [0.51718, 0.54539, 0.57359, 0.60180],
                        [0.52321, 0.55141, 0.57962, 0.60782],
                ],
        ]
).to(torch.float32)

print("scaled_dot_product_no_loop_batch error: ", rel_error(y_expected, y))
```

```
scaled_dot_product_no_loop_batch error: 4.020571992067902e-06
```

使用雙層迴圈和不使用迴圈的時間比較，發現差了 4 倍。

```
N = 64
K = 256   # defines the input sequence length
M = emb_size = 2048
dim_q = dim_k = 2048
query = torch.linspace(-0.4, 0.6, steps=N * K * M).reshape(N, K, M)    # **to_double_cuda
key = torch.linspace(-0.8, 0.5, steps=N * K * M).reshape(N, K, M)    # **to_double_cuda
value = torch.linspace(-0.3, 0.8, steps=N * K * M).reshape(N, K, M)    # *to_double_cuda

%timeit -n 5 -r 2   y = scaled_dot_product_no_loop_batch(query, key, value)
```

```
763 ms ± 143 ms per loop (mean ± std. dev. of 2 runs, 5 loops each)
```

```
N = 64
K = 512   # defines the input requence length
M = emb_size = 2048
dim_q = dim_k = 2048
query = torch.linspace(-0.4, 0.6, steps=N * K * M).reshape(N, K, M)    # **to_double_cuda
key = torch.linspace(-0.8, 0.5, steps=N * K * M).reshape(N, K, M)    # **to_double_cuda
value = torch.linspace(-0.3, 0.8, steps=N * K * M).reshape(N, K, M)    # *to_double_cuda

%timeit -n 5 -r 2   y = scaled_dot_product_no_loop_batch(query, key, value)
```

```
2.52 s ± 41.8 ms per loop (mean ± std. dev. of 2 runs, 5 loops each)
```

四. SelfAttention:

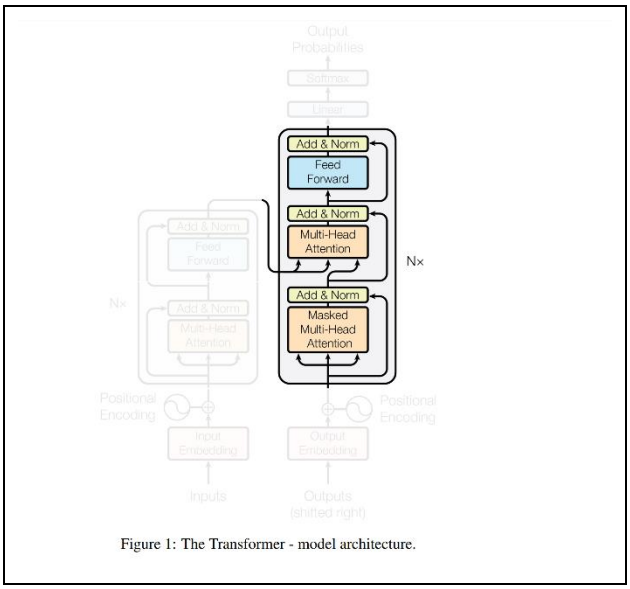


Figure 1: The Transformer - model architecture.

這裡主要是用 class 的方式完成 self attention。

__init__主要是將輸入透過 nn.linear 把 q k v 初始化，並將 q k v 的權重初始為一個 uniform 的機率分佈，大小為[-c , c]，c = $\sqrt{\frac{6}{Din+Dout}}$。

程式碼如下:

```
self.q=nn.Linear(dim_in,dim_q)
c=math.sqrt(6/(dim_in+dim_q))
torch.nn.init.uniform_(self.q.weight.data,a=-c,b=c)

self.k=nn.Linear(dim_in,dim_q)
c=math.sqrt(6/(dim_in+dim_q))
torch.nn.init.uniform_(self.k.weight.data,a=-c,b=c)

self.v=nn.Linear(dim_in,dim_v)
c=math.sqrt(6/(dim_in+dim_v))
torch.nn.init.uniform_(self.v.weight.data,a=-c,b=c)
```

Forward 主要將 q k v 進行維度轉換，並把他帶入 scaled dot product no loop batch 裡面，得到輸出和 softmax 權重矩陣。

程式碼如下:

```
Q=self.q(query)
K=self.k(key)
V=self.v(value)
y,self.weights_softmax=scaled_dot_product_no_loop_batch(Q,K,V,mask)
```

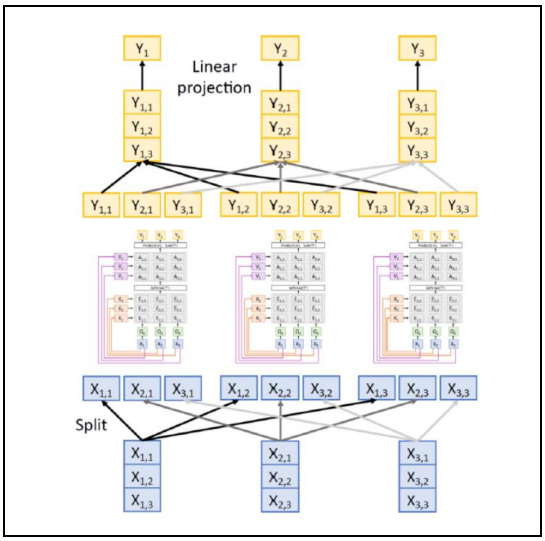
執行結果如下:

```
print("SelfAttention error: ", rel_error(y_expected, y))
print("SelfAttention error: ", rel_error(dy_expected, query_grad))
```

```
SelfAttention error:  5.282987963847609e-07
SelfAttention error:  2.474069076879365e-06
```

五. MultiHeadAttention:

MultiHeadAttention 的概念如下圖所示。主要目的為，讓模型能夠並行地學習不同的關注方向和特徵表示。每個 attention 頭可以專注於序列中的不同位置、不同的語義層面或不同的注意力模式。這樣的並行計算增加了模型的表達能力，有助於捕捉更多的序列關聯性和特徵信息。:



__init__的定義如下:

1. self.num_head : attention 機制裡面頭的數量。

2. self.attns ： 根據 num_head 個數，用 nn.modulelist 把不同的頭分配到不同的 self attention 的 block 進行運算
3. self.linear_proj ： 運算完後，用 nn.linear 把剛剛分散的頭變成原本輸入的形式。

程式碼如下:

```
self.num_heads = num_heads

self.attns = nn.ModuleList([SelfAttention(dim_in, dim_out, dim_out) for _ in range(num_heads)])
self.linear_proj = nn.Linear(num_heads * dim_out, dim_in)
```

forward 的定義如下:

1. 先把 q k v mask 代入 self.attns，分散到不同的 block 計算，並用一個 list 存起來。
2. 把算完的結果用 torch.cat 的方式串起來，變成一個 tensor。
3. 用 self.linear_proj 把結果變成原本輸入的形式。

程式如下:

```
y=[h(query, key, value, mask) for h in self.attns]
y=torch.cat(y,dim=-1)
y = self.linear_proj(y)
```

執行結果如下:

```
MultiHeadAttention error:  5.366163452092416e-07
MultiHeadAttention error:  6.127381394302327e-07
```

六. LayerNormalization:

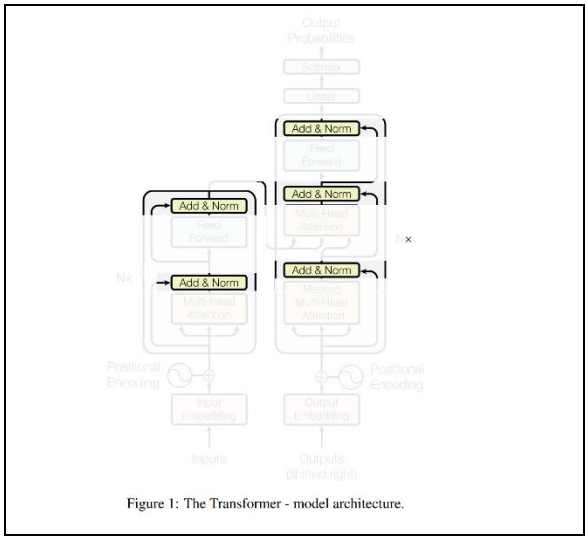


Figure 1: The Transformer - model architecture.

這裡的目的是將輸入的特徵在通道維度上進行歸一化，以減少不同特徵之間的相互依賴性，從而有助於提高模型的穩定性和學習能力。

Normalization 的公式如下:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

__init__的定義如下:

1. self.shift : 用 nn.Parameter 定義了一個可學習的參數 shift，形狀為(emb_dim,)，初始值為全 1。scale 用於對輸入進行偏移。
2. self.scale : 用 nn.Parameter 定義了一個可學習的參數 scale，形狀為(emb_dim,)，初始值為全 1。scale 用於對輸入進行線性縮放操作。

程式碼如下:

```
self.scale=nn.Parameter(torch.ones(emb_dim))
self.shift=nn.Parameter(torch.zeros(emb_dim))
```

Forward 主要是把 x 帶入 normalization 的公式，程式碼如下:

```
y=(self.scale*(x-x.mean(dim=-1,keepdim=True))/(x.std(dim=-1,keepdim=True,unbiased=False)+self.epsilon))+self.shift
```

執行結果:

```
print("LayerNormalization error: ", rel_error(y_expected, y))
print("LayerNormalization error: ", rel_error(dy_expected, inp_grad))
```

```
LayerNormalization error:  1.3772273765080196e-06
LayerNormalization error:  2.2667419499750657e-07
```

七. FeedForward:

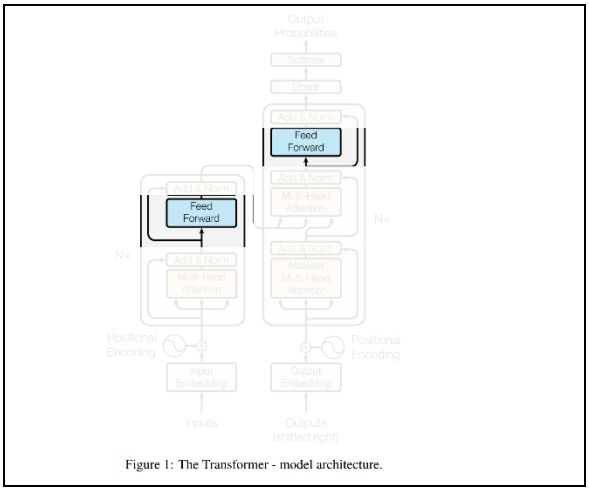


Figure 1: The Transformer - model architecture.

FeedForward 輸入為 MultiAttention 的輸出，主要的成分為兩個 MLP 和兩個 ReLU。

__init__的定義如下:

1. self.linear1 : 為第一個 MLP 層，並將此 MLP 層的權重用 xavier 初始化。
2. self.linear2 : 為第二個 MLP 層，並將此 MLP 層的權重用 xavier 初始化。
3. self.relu: ReLU 層。

程式碼如下:

```
self.linear1=nn.Linear(inp_dim,hidden_dim_feedforward)
torch.nn.init.xavier_uniform_(self.linear1.weight.data)
self.relu=nn.ReLU()
self.linear2=nn.Linear(hidden_dim_feedforward,inp_dim)
torch.nn.init.xavier_uniform_(self.linear2.weight.data)
```

Forward 主要的順序為 x->linear1->relu->linear2，程式碼如下:

```
y=self.linear2(self.relu(self.linear1(x)))
```

程式執行結果:

```
print("FeedForwardBlock error: ", rel_error(y_expected, y))
print("FeedForwardBlock error: ", rel_error(dy_expected, inp_grad))

FeedForwardBlock error:  2.1976864847460601e-07
FeedForwardBlock error:  2.302209886634859e-06
```

八. EncoderBlock:

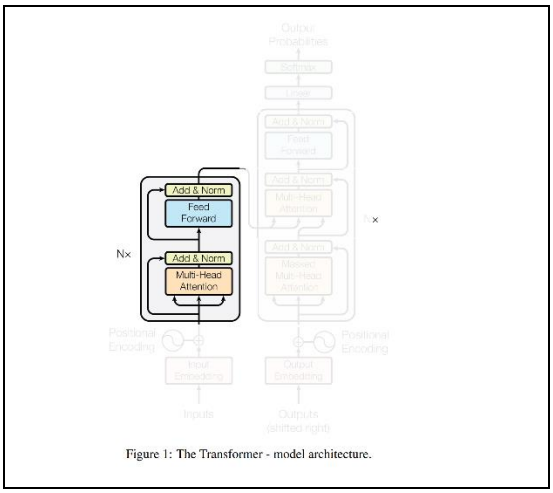


Figure 1: The Transformer - model architecture.

這裡主要是把剛剛寫的 MultiHeadAttention 和 LayerNormalization 和 FeedForward 組合成一個 block，為 Transformer 的 encoder 部分。

__init__的定義如下:

1. self.mult_attn : 定義了 MultiHeadAttention 層。
2. self.norm1 : 為 Encoder 下面的 LayerNormalization 層。
3. self.norm2 : 為 Encoder 上面的 LayerNormalization 層。
4. self.feed : 定義了 FeedForward 層。
5. self.drop : 定義了 dropout

程式碼如下:

```
self.mult_attn=MultiHeadAttention(num_heads,emb_dim,emb_dim//num_heads)
self.norm1=LayerNormalization(emb_dim)
self.norm2=LayerNormalization(emb_dim)
self.feed=FeedForwardBlock(emb_dim,feedforward_dim)
self.dropout=nn.Dropout(dropout)
```

forward 的部分就按照 EncoderBlock 的定義代入不同的式子。比較不一樣的是，當每一次 Add_Norm 之後要加上 dropout。程式如下:

```
x=self.dropout(self.norm1(x+self.mult_attn(x,x,x)))
y=self.dropout(self.norm2(x+self.feed(x)))
```

執行結果如下:

```
encoder_out2 = enc_block(enc_seq_inp)
print("EncoderBlock error 2: ", rel_error(encoder_out2, encoder_out2_expected))
```

```
EncoderBlock error 1:  5.737253709511974e-07
EncoderBlock error 2:  5.58627368610239e-07
```

九. get_subsequent_mask:

為了製造 mask，所以我們要建立一個下三角矩陣，下面為 False，上面為 True。shape 為(N,K,K)，N 為 batch size，k 為 sequence length。程式如下:

```
N, K = seq.shape
mask = (1 - torch.tril(torch.ones((N, K, K), device=seq.device))).bool()
```

執行結果如下:

```
mask_predicted = get_subsequent_mask(inp_sequence)
print(
    "get_subsequent_mask error: ", rel_error(mask_predicted.int(), mask_expected.int())
)
```

```
get_subsequent_mask error:  0.0
```

看到之前寫的 scaled_dot_product_no_loop_batch，當 mask 不等於 None 時，結果是對的。

```
y_predicted, _ = scaled_dot_product_no_loop_batch(query, key, value, mask_expected)

print("scaled_dot_product_no_loop_batch error: ", rel_error(y_expected, y_predicted))
```

```
scaled_dot_product_no_loop_batch error:  2.8390648478191238e-06
```

十. DecoderBlock:

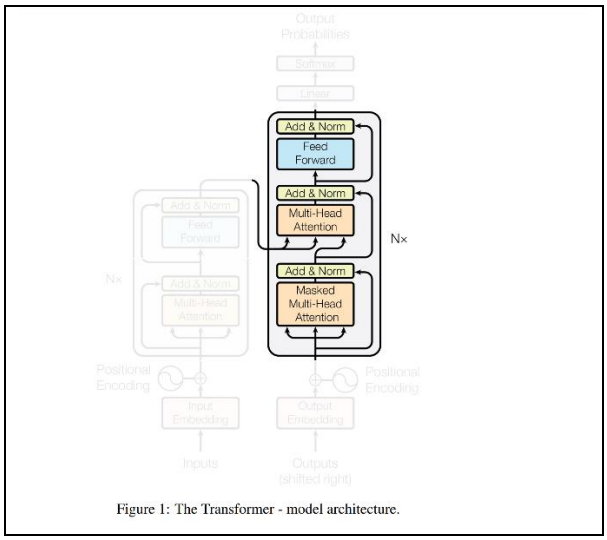


Figure 1: The Transformer - model architecture.

這裡主要是把剛剛寫的 MultiHeadAttention 和 LayerNormalization 和 FeedForward 和 MaskMultiHeadAttention 組合成一個 block，為 Transformer 的 encoder 部分。

__init__的定義如下:

1. self.attention_self ： 定義了 Decoder 下面的 MultiHeadAttention 層。
2. self.attention_cross ： 定義了 Decoder 中間的 MultiHeadAttention 層。
3. self.norm1 ： 為 Decoder 下面的 LayerNormalization 層。
4. self.norm2 ： 為 Decoder 中面的 LayerNormalization 層。
5. self.norm3 ： 為 Decoder 上面的 LayerNormalization 層。
6. self.feed_forward ： 定義了 Decoder 上面的 FeedForward 層。
7. self.drop ： 定義了 dropout

程式碼如下:

```
self.attention_self  =  MultiHeadAttention(num_heads,emb_dim,emb_dim//num_heads)
self.attention_cross  =  MultiHeadAttention(num_heads,emb_dim,emb_dim//num_heads)
self.feed_forward  =  FeedForwardBlock(emb_dim,feedforward_dim)
self.norm1  =  LayerNormalization(emb_dim)
self.norm2  =  LayerNormalization(emb_dim)
self.norm3  =  LayerNormalization(emb_dim)
self.dropout  =  nn.Dropout(dropout)
```

forward 的部分就按照 DecoderBlock 的定義代入不同的式子。程式如下:

```
#下
x=self.dropout(self.norm1(dec_inp+self.attention_self(dec_inp,dec_inp,dec_inp,mask)))
#中
x=self.dropout(self.norm2(x+self.attention_cross(x,enc_inp,enc_inp)))
#上
y=self.dropout(self.norm3(x+self.feed_forward(x)))
```

程式執行結果:

```
dec_out2  =  dec_block(dec_inp,  enc_out)
print("DecoderBlock  error:  ",  rel_error(dec_out2,  dec_out_expected))

DecoderBlock error:  0.49748849183075144
DecoderBlock error:  0.5007399975709957
```

Part III. Data loader

一. position_encoding_simple:

實現了一個簡單的位置編碼，用於生成一個序列的位置編碼表示。

1. 用 torch.linspace(0,1-1/K,steps=K)生成一個長度為 K 的 Tensor，K 為序列長度，其中每個元素的值為 n/K，n 從 0 開始遞增。這樣的數列可以表示序列中每個位置的相對位置。
2. 用 repeat 的方式把 shape 變成(1,M,K)，M 為位置編碼的長度，再用 permute 函式將維度重新排列為(1,K,M)。

程式碼如下:

```
y=torch.linspace(0,1-1/K,steps=K)
y=y.repeat((1,M,1)).permute(0,2,1)
```

執行結果如下:

```
print("position_encoding_simple  error:  ",  rel_error(y,  y_expected))

position_encoding_simple error:  0.0
position_encoding_simple error:  0.0
```

二. Sinusoid positional encoding:

這裡實現了一個正弦型（sinusoidal）的位置編碼函式 position encoding sinusoid，用於生成序列的位置編碼表示。公式如下:

$$PE_{(p,2i)} = \sin\left(\frac{p}{10000^a}\right)$$
$$PE_{(p,2i+1)} = \cos\left(\frac{p}{10000^a}\right)$$
$$\text{Where } a = \left\lfloor \frac{2i}{M} \right\rfloor \text{ and M is the Embedding dimension of the Transformer}$$

1. 用 pos=torch.arange(K,dtype=torch.float).reshape(1,-1,1)生成一個形狀為(1,K,1) 的 Tensor，其中每個元素的值為從 0 到 K-1 的連續整數，表示序列中每個位置的索引。
2. 用 dim=torch.arange(M,dtype=torch.float).reshape(1,1,-1)生成一個形狀為(1,1,M) 的 Tensor，其中每個元素的值為從 0 到 M-1 的連續整數，表示序列中位置編碼的維度。
3. 利用 phase=pos/(10000**(torch.div(dim,M,rounding_mode='floor'))，根據給定的公式計算位置編碼的相位值。torch.div(dim, M, rounding_mode='floor') 表示將 dim 除以 M，並向下取整數。
4. 利用 torch.where，根據位置編碼的嵌入維度 dim 的奇偶性，選擇使用 sin 或 cos 來計算位置編碼的值。如果 dim 是偶數，則使用 sin(phase)；如果 dim 是奇數，則使用 cos(phase)。

程式碼如下:

```
pos=torch.arange(K,dtype=torch.float).reshape(1,-1,1)
dim=torch.arange(M,dtype=torch.float).reshape(1,1,-1)
phase=pos/(10000**(torch.div(dim,M,rounding_mode='floor')))
y=torch.where(dim.long()%2==0,torch.sin(phase),torch.cos(phase))
```

執行結果如下:

```
position_encoding error:  1.5795230865478516e-06
position_encoding error:  1.817941665649414e-06
```

Part IV: Using transformer on the toy dataset

一.Transformer:

__init__需要寫的部分為將輸入字串丟入 nn.Embedding 把字串變成對應的向量。程式碼如下:

```
self.emb_layer = nn.Embedding(vocab_len, emb_dim)
```

Forward 的主要寫的部分如下:

1. 將輸入的問題向量 q_emb_inp 作為輸入，通過編碼器進行處理，獲得輸出 enc_out。
2. 把 ans_b 代入 get_subsequent_mask 生成一個 mask，用於在解碼器中遮蔽未來時間的資訊。ans_b 是答案序列的 tensor，shape 為(batch_size, seq_length)，ans_b[:, :-1]即去掉最後一個時間。這是因為在解碼器中，我們的目標是預測下一個時間的答案，所以不需要考慮最後一個時間的資訊。
3. 將輸入的答案向量 a_emb_inp、編碼器的輸出 enc_out 和 mask 作為輸入，通過解碼器進行處理，獲得解碼器的輸出 dec_out。
4. 將 dec_out 換為一個 2D tensor，目的是為了方便進行後續的計算和處理。

程式碼如下:

```
enc_out=self.encoder(q_emb_inp)
mask=get_subsequent_mask(ans_b[:,:-1])
dec_out=self.decoder(a_emb_inp,enc_out,mask)
dec_out = dec_out.view(-1, dec_out.size(-1))
```

二.Transformer 執行結果:

1. 利用 Transformer 把少量 data train 成 overfitting，發現最後準確率為 1。

```
[epoch: 193] [loss:  0.1633 ] val_loss: [val_loss  0.0728 ]
[epoch: 194] [loss:  0.1835 ] val_loss: [val_loss  0.0743 ]
[epoch: 195] [loss:  0.2305 ] val_loss: [val_loss  0.0754 ]
[epoch: 196] [loss:  0.1156 ] val_loss: [val_loss  0.0768 ]
[epoch: 197] [loss:  0.1897 ] val_loss: [val_loss  0.0760 ]
[epoch: 198] [loss:  0.1532 ] val_loss: [val_loss  0.0742 ]
[epoch: 199] [loss:  0.2813 ] val_loss: [val_loss  0.0717 ]
[epoch: 200] [loss:  0.1864 ] val_loss: [val_loss  0.0706 ]
```

```
#Overfitted  accuracy
print(
        "Overfitted  accuracy:  ",
        "{:.4f}".format(
                val_transformer(
                        trained_model,
                        small_train_loader,
                        CrossEntropyLoss,
                        batch_size=4,
                        device=DEVICE,
                )[1]
        ),
)
```

```
Overfitted accuracy:  1.0000
```

2. 利用 transformer 訓練完整 data，花了 200 個 epoch，end_dim 和 feedforward_dim 都是 128，最後準確率落在 80.57%

```
[epoch: 190] [loss:  0.4883 ] val_loss: [val_loss  0.4196 ]
[epoch: 191] [loss:  0.4845 ] val_loss: [val_loss  0.3990 ]
[epoch: 192] [loss:  0.4899 ] val_loss: [val_loss  0.3998 ]
[epoch: 193] [loss:  0.4801 ] val_loss: [val_loss  0.3894 ]
[epoch: 194] [loss:  0.4726 ] val_loss: [val_loss  0.3872 ]
[epoch: 195] [loss:  0.4707 ] val_loss: [val_loss  0.3980 ]
[epoch: 196] [loss:  0.4712 ] val_loss: [val_loss  0.4024 ]
[epoch: 197] [loss:  0.4611 ] val_loss: [val_loss  0.3729 ]
[epoch: 198] [loss:  0.4714 ] val_loss: [val_loss  0.3969 ]
[epoch: 199] [loss:  0.4672 ] val_loss: [val_loss  0.4079 ]
[epoch: 200] [loss:  0.4836 ] val_loss: [val_loss  0.4522 ]
```

```
[47] #Final  validation  accuracy
     print(
             "Final  Model  accuracy:  ",
             "{:.4f}".format(
                     val_transformer(
                             trained_model,  valid_loader,  LabelSmoothingLoss,  4,  device=DEVICE
                     )[1]
             ),
     )
```

```
     Final Model accuracy:  0.8057
```