深度學習

HW12

學號 : B103012002

姓名 : 林凡皓

一、 Load the toy data and Preprocessing data

Transformers 對於 seq-to-seq 的資料有很好的表現，這一次作業中我們將它用來做數學運算。這次作業會採用 transformer 模型來算數學的加減法。

首先查看這次要訓練的資料，前四筆資料如下

```
Expression: BOS NEGATIVE 30 subtract NEGATIVE 34 EOS Output: BOS POSITIVE 04 EOS
Expression: BOS NEGATIVE 34 add NEGATIVE 15 EOS Output: BOS NEGATIVE 49 EOS
Expression: BOS NEGATIVE 28 add NEGATIVE 36 EOS Output: BOS NEGATIVE 64 EOS
Expression: BOS POSITIVE 00 subtract POSITIVE 17 EOS Output: BOS NEGATIVE 17 EOS
```

以第一個資料為例作說明。

第一筆資料的輸入為 BOS NEGATIVE 30 subtract NEGATIVE 34 EOS，輸出為 BOS POSITIVE 04 EOS。將它翻譯成我們習慣的數學表示式就會變成輸入(-30) – (-34)，得到輸出+4，而 BOS 和 EOS 分別代表序列的開始與結束。

接著對這些數據做 tokenization。Tokenization 為一種資料預處理方式，我們需要先將人類閱讀的文字轉換為一個序列的 token。首先我們需要創建一個詞彙表，在這次作業中，詞彙表會由 16 個元素組成，分別為數字 0~9，正負號標記(POSITIVE、NEGATIVE)、運算標記(add、subtract)和序列開始與結束的標記(BOS、EOS)。

通常我們以 list 來實現詞彙表，其中 vocab[i] = s 代表說字串符號 s 被分配到索引 i。此外，我們會建立一個 dictionary convert_str_to_token 來幫助我們將字串映射到其相應的整數索引。

建立好 vocab 之後我們實現兩個函數分別為 generate_token_dict 和 preprocess_input_sequence。

- generate_token_dict :
  1. 實現方法 ：

     此函數的功能為接收一個 list vocab 並回傳一個 dictionary，此 dictionary 為用來 convert_str_to_token 的字典。

     透過 enumerate(vocab)來抓取 vocab list 中的索引與存放的單字，定將抓取到的索引設定為 dictionary 的數值，單字設定為 dictionary 的 key。
  2. 執行結果 ：

     ```
     Dictionary created successfully!
     ```

- preprocess_input_sequence :

  1. 實現方法 :

     此函數用來將一個輸入字串轉換為 vocab list 中的 token。

     首先要利用 split 將輸入句子中每個字都分開，並對每一個字做迭代。每一次迭代先去判斷目前的字為特殊字元或是數字，如果是特殊字元就利用 dictionary 將 token 抓出來，如果是數字就要針對每一個數字再做一次迭代(因為最初的數字可能為二位數)，再去 dictionary 中抓取 token。

  2. 執行結果 :

```
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
```

二、 Implementing transformer building blocks

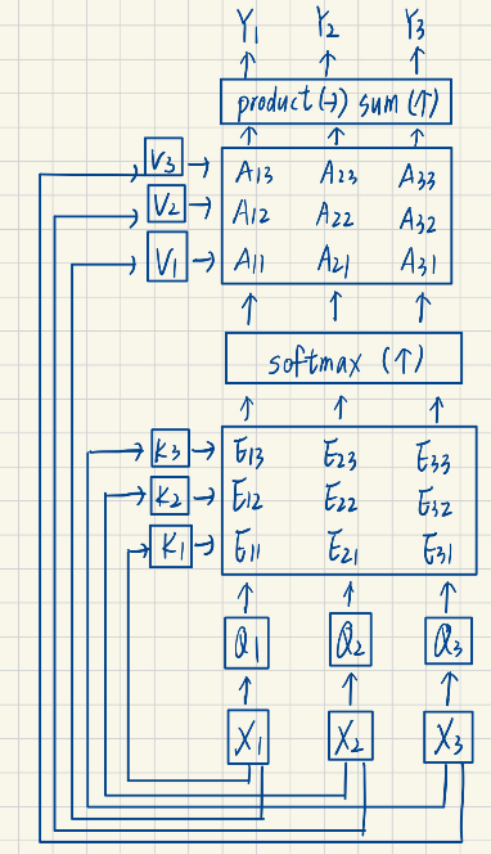
完成資料預處理後，接下來要實作 transformer 使用的 building block，而這些 building block 會在後續建構 encoder 與 decoder 時用到。

這邊先實作 MultiHeadAttention block、FeedForward block、Layer Normalization 和 Positional Encoding block。

在實作 MultiHeadAttentin block 之前，由於我們想透過對 Self Attention block 做擴充的方式來實現 MultiHeadAttention block，因此先去實作 Self Attention block。

- Self Attention block :

  Self attention block 的架構如下



根據此架構，實現 scaled_dot_product_two_loop_single、scaled_dot_product_two_loop_batch 和 scaled_dot_product_no_loop_batch。

1. scaled_dot_product_two_loop_single :

   此函數為最基本的 self attention block。

   利用兩個 for loop 計算 query 和 key 的乘積，每一次迭代都去計算 query 與 key 的內積。接著透過$E = \frac{QK^T}{\sqrt{M}}$來計算 attention。再將 attention 送入 softmax 後，與 value 相乘得到最後的輸出。

   檢查結果如下

   ```
   sacled_dot_product_two_loop_single error:  5.2049970024350008e-06
   ```

2. scaled_dot_product_two_loop_batch :

   根據剛才實現的 scaled_dot_product_two_loop_simple 做延伸，將它改成對 batch N 的 self attention。與上一個函數的主要差別在於剛才每次迭代的內積修改為 torch.einsum 來實現對每個 batch 做內積。

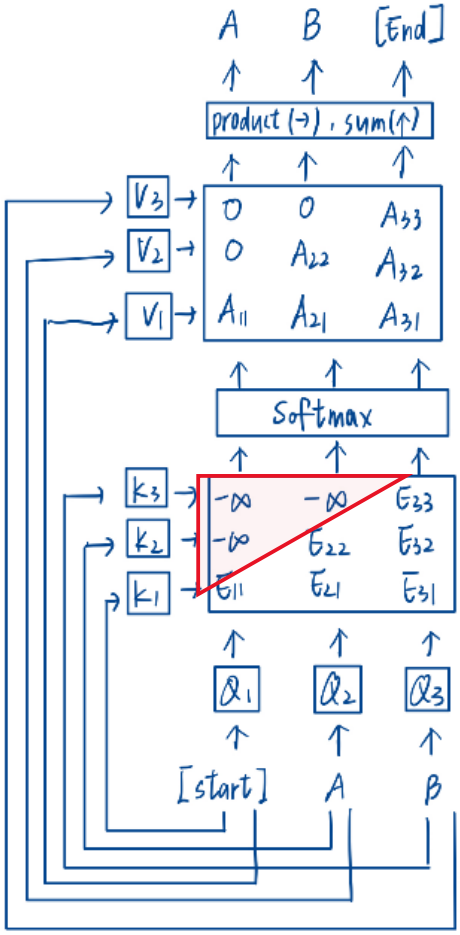
   檢查結果如下

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

3. scaled_dot_product_no_loop_batch :

此函數有加入一個 mask 的選項，用來選擇是否要採用 masked self attention。在 mask = False 時，此函數做的事與剛才的 scaled_dot_product_two_loop_batch 相同，差別在於這邊直接使用 torch.bmm 來實現 query 和 key 在每個 batch 的相乘，而非透過迴圈。

當 mask = True 時，主要的差別在於 attention 的部分，詳細 masked self attention 的架構如下



由上圖可以注意到，masked self attention 的 attention 會有部分被設定為負無窮大的數值(紅色三角形)，這主要是因為我們希望忽略掉這些位置的影響。在實作的部分，我們則利用-1e-9 來取代負無窮大。

檢查結果如下

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

實作完成後，討論 transformer 的時間複雜度。分別對 sequence

length = 256 與 sequence length = 512 的序列做測試，測試結果如下

sequence length = 256 :

```
63.1 ms ± 8.07 ms per loop (mean ± std. dev. of 2 runs, 5 loops each)
```

sequence length = 512 :

```
254 ms ± 16.5 ms per loop (mean ± std. dev. of 2 runs, 5 loops each)
```

由結果可以看到時間差了四倍，因此 transformer 的時間複雜度約為$O(n^2)$。

接著將 SelfAttention block 完成。

1. __init__ :

   初始化部分有 q、k 和 v 要做初始化，channel shape 分別為 (dim_in, dim_q)、(dim_in, dim_q)、(dim_in, dim_v)。這變將他們初始化為一個 linear layer，因為它們主要用來將輸入映射到 query、key 和 value。關於 linear layer 中權重的初始化，weight 都是初始化為大小介於 $-\sqrt{\frac{6}{D_{in}+D\_out}}$ 到 $\sqrt{\frac{6}{D_{in}+D\_out}}$ 之間的 uniform distribution，而 bias 則初始化為 0。

2. forward :

   forward path 的部分即剛才定義好的 scaled_dot_product_no_loop_batch。

3. 執行結果 :

```
SelfAttention error:  5.2829879638476O9e-07
SelfAttention error:  3.0810741803438454e-06
```

- MultiHeadAttention:

  MultiHeadAttention block 將輸入分為多個小塊，每一塊會有一個 self attention block 去處理，以達到平行計算的效果，最後再將每一個 self attention 的輸出結果拼接起來，形成最終輸出。

  1. 實現方法 :

     初始化部分，由於 MultiHeadAttention block 有多個 SelfAttention block，因此可以透過 nn.ModuleList 來實現多個 SelfAttention block 的架構，而 SelfAttention block 的數量取決於 heads 的數量。此外我們還需要一層 linear layer 將每一個 SelfAttention block 的輸出結果轉換為輸入維度。關於 linear layer 權重初始化部分，

weight 採用 xavier 的初始化方式，而 bias 初始化為 0。

Forward path 的部分，將 query、key、value 送進每一個 SelfAttention block 計算出輸出，再將所有輸出拼起來，最後再透過 linear layer 將維度映射到輸入的維度。

2. 執行結果 ：

```
MultiHeadAttention error:  5.366163452092416e-07
MultiHeadAttention error:  1.2122403342308667e-06
```

- LayerNormalization :

在做 CNN 相關作業時有時做過 Batch normalization，但是 batch normalization 會因為批次的大小而影響到表現，特別是在處理序列資料時，因為序列長度不同導致批次大小不穩定，batch normalization 的表現也會跟著不穩定，因此這次改為使用 layer normalization。Layer normalization 不依賴批次大小，而是對每個 timestep 單獨做歸一化，其優點在於它可以做平行化處理，而且在測試時的運作方式與在訓練時相同。

1. 實現方法 ：

首先是初始化部分，這邊要初始化 scale 和 shift，scale 初始化為 1，shift 初始化為 0。

Forward path 部分即根據 layer normalization 的公式做實現，公式如下

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \odot \gamma + \beta$$

公式中 μ 為 x 的 mean、σ 為 x 的標準差、ϵ 惟為了防止分母為零而加入的一個微小常數、γ 為 scale、β 為 shift。

2. 執行結果 ：

```
LayerNormalization error:  1.3772273765080196e-06
LayerNormalization error:  2.6788768499705324e-07
```

- FeedForwardBlock :

Feed forward block 會在 transformer 的 encoder 和 decoder 中用到，通常由堆疊多層的 MLP 與 ReLU 所構成。

1. 實現方法 ：

初始化部分，初始化兩個 MLP，及 linear layer → ReLU → linear layer。Shape 的部分輸入和輸出的形狀相同。參數初始化部分，

weight 都利用 xavier initialization，bias 都初始化為 0。

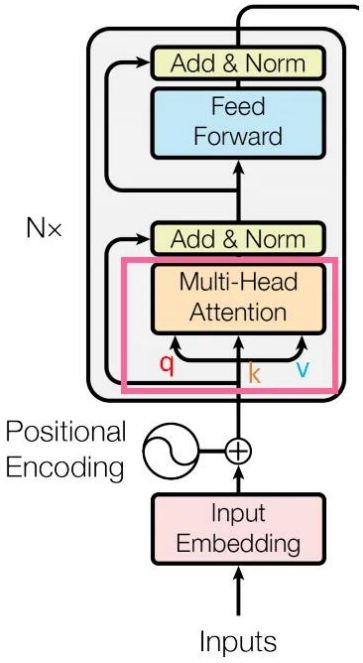
Forward path 部分只需將輸入送進 linear layer 1 → ReLU → linear layer 2 即可。

2. 執行結果 ：

```
FeedForwardBlock error:  2.1976864847460601e-07
FeedForwardBlock error:  2.302209744348415e-06
```

- EncoderBlock :

Encoder block 由 MultiHeadAttention、FeedForward、LayerNormalization 組成。在 transformer 的 encoder 中，每個 encoder 會接收三個輸入，分別為 query、key、value。Encoder 架構如下



關於 positional encoding 的部分，由於他是不可學習的，因此這邊將他視作 data loader 中預處理的步驟。

接著根據上面架構建立 encoder。

1. 實現方法 ：

利用先前建立好的 MultiHeadAttention、LayerNormalization、FeedForwardBlock 來建構 encoder。初始化部分只要設定好這些 layer 即可。

Forward path 部分，要特別注意的是 residual 的部分。第一層 LayerNormalization 的輸入為做一開始的輸入 x，加上經過 MultiHeadAttention 後的結果。第二層 LayerNormalization 的輸入

也是 residual 後的結果。

2. 執行結果 ：

```
EncoderBlock error 1:  0.08512793741417692
EncoderBlock error 2:  0.5019787982258388
```

- DecoderBlock :

在比較複雜的任務中，我們會需要 decoder 來將 encoder 的輸出轉換為我們想要的序列樣式。Decoder 會接收 encoder 的輸出以及前一個產生的結果來產生下一個結果。在訓練過程中，我們會對輸入使用 mask，以防止 decoder 提前看到未來的輸入，在預測的時候則是循序的處理資料。

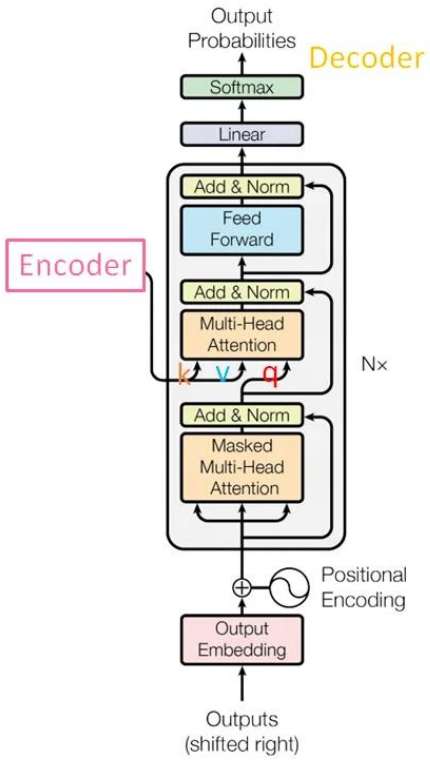
1. Get_subseqeunt_mask :

Mask 主要是為了避免 decoder 提前得知未來資訊。根據 masked self attention 的架構可以得知，mask 應該為一個上三角矩陣，因此這邊先去產生一個全部為 1 的矩陣，再產生出一個全部元素為 1 的下三角矩陣，將兩個相減得到全部為 1 的上三角矩，再將矩陣中數值轉為布林值。測試結果如下

```
get_subsequent_mask error:  0.0
```

2. Decoder 實現方法 ：

Decoder 架構如下



初始化部分需要初始化兩個 MultiHeadAttention、三個

LayerNormalization 和一個 FeedForwardBlock。

Forward path 部分根據架構圖實現，要特別注意的是第一個 MultiHeadAttention 為 masked self attention。

3. Decoder 檢查結果 ：

```
DecoderBlock error:  0.4999456863965454
DecoderBlock error:  0.5031966572696958
```

三、 Data loader

- Simple positional encoding :

  由於 transformer 中沒有內建的位置資訊，因此需要透過 positional encoding 來未輸入數據加上位置資訊。這些位置資訊通常會與輸入做相加，因此形狀應該與輸入相同。這邊時做最簡單的 positional encoding，即將序列中第 n 個元素分配給 n/k 的值。

  1. 實現方法 ：

     透過 torch.linspace 產生出 0 ~ 1 – 1/k，等間距的張量，再透過 repeat 的方式將其擴展以及 permute 讓形狀符合輸入的形狀。

  2. 檢查結果 ：

```
position_encoding_simple error:  0.0
position_encoding_simple error:  0.0
```

- Sinusoid positional encoding :

  Simple positional encoding 的缺點在於當序列長度增加時，兩個連續位置編碼的差異會變很小，導致編碼的作用下降。因此改為利用 sin、cos 來做 positional encoding。

  Sinusoid positional encoding 的公式如下

$$PE_{(p,2i)} = \sin\left(\frac{p}{10000^a}\right)$$
$$PE_{(p,2i+1)} = \cos\left(\frac{p}{10000^a}\right)$$

  其中 $a = \frac{2i}{M}$ ，M 為 transformer 的 embedding dimension。

  1. 實現方法 ：

     根據上述公式完成即可。

  2. 檢查結果 ：

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

四、 Using transformer on the toy dataset

- Implement the transformer model :

  初始化部分我們只需要實作 embedding layer 的部分。Embedding layer 可以透過呼叫 torch.nn 內建的 embedding layer 來實現，其主要功能為將 vocab_len 映射到 emb_dim。

  Forward path 的部分，首先將 q_emb_inp 送進 encoder，接著透過先前建立好的 get_subsequent_mask 來建構 mask，並將 encoder 的 ouput、a_emb_inp 和 mask 一起送進 decoder 中。

- Overfitting the model using small data :

  使用小型資料嘗試讓模型 overfit，以確認模型能夠正常作訓練。訓練 200 個 epochs 後結果如下

```
[epoch: 196] [loss:  0.1115 ] val_loss: [val_loss  0.0173 ]
[epoch: 197] [loss:  0.1090 ] val_loss: [val_loss  0.0169 ]
[epoch: 198] [loss:  0.0654 ] val_loss: [val_loss  0.0165 ]
[epoch: 199] [loss:  0.1056 ] val_loss: [val_loss  0.0162 ]
[epoch: 200] [loss:  0.0972 ] val_loss: [val_loss  0.0161 ]
```

```
Overfitted accuracy:  1.0000
```

  最後準確度為 1，表示模型有順利 overfit。

- Fit the model using complete data :

  利用完整資料及來訓練模型。這部分由於作業建構的模型表現不是很好，因此我嘗試修改一些參數來提升模型表現。

  首先我修改模型架構，將 embedding dimension 與 feedforward dimension 提升到 256、512，number of head 提升至 8，number of encoder layer 與 number of decoder layer 設定為 4。

  此外我也將 positional encoding 改為使用 sinusoid positional encoding。

  訓練 300 個 epochs 後結果如下

```
[epoch: 296] [loss:  0.9365 ] val_loss: [val_loss  0.9007 ]
[epoch: 297] [loss:  0.9360 ] val_loss: [val_loss  0.8937 ]
[epoch: 298] [loss:  0.9407 ] val_loss: [val_loss  0.9047 ]
[epoch: 299] [loss:  0.9388 ] val_loss: [val_loss  0.9359 ]
[epoch: 300] [loss:  0.9430 ] val_loss: [val_loss  0.8724 ]
```

```
Final Model accuracy:  0.8516
```

  最終模型有 85.16 %的準確度。

- Visualize and inference : model in action

  將剛才訓練完成的模型拿來作預測，部分結果如下

```
Input sequence:
 BOS NEGATIVE 07 add NEGATIVE 12 EOS
Output Sequence:          BOS NEGATIVE 1 9

Input sequence:
 BOS NEGATIVE 31 add POSITIVE 13 EOS
Output Sequence:          BOS NEGATIVE 1 8

Input sequence:
 BOS POSITIVE 26 add NEGATIVE 33 EOS
Output Sequence:          BOS NEGATIVE 0 7
```

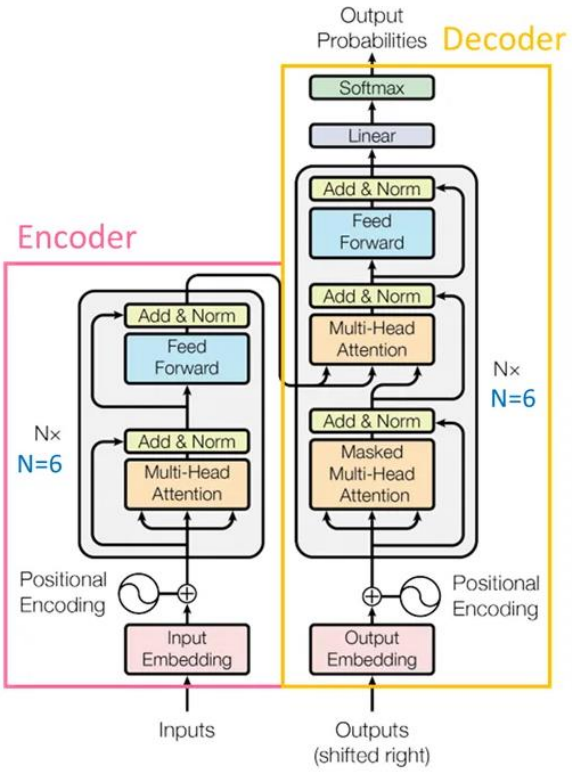
可以看到模型是能夠正確計算出結果。

五、 額外嘗試

- Attention is all you need :

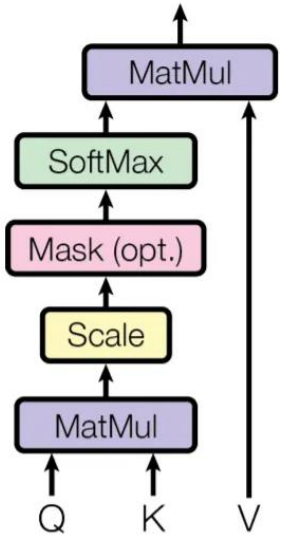
這次作業基本上都圍繞在"Attention is all you need"這篇論文，因此這邊將這篇論文內容做簡單的整理。

Transformer 架構主要由 encoder、decoder 堆疊而成，在論文中，作者使用六層的 encoder 與 decoder 來實現，encoder-decoder 架構如下。



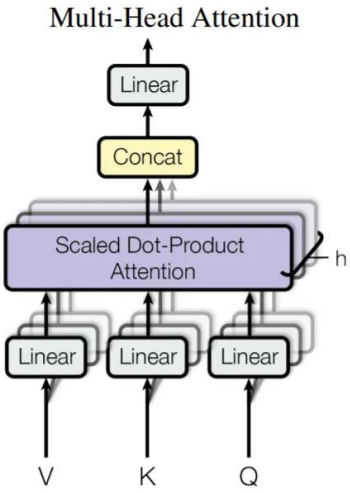
首先看到 encoder 的部分，輸入會先經過 embedding 轉換為向量，接著再與 positional encoding 做相加以涵蓋位置資訊。Positional encoding 部分，論文採用 sinusoid encoding 的方式做編碼，公式部分已在實作 sinusoid positional encoding 時說明過，這邊主要討論為甚麼要用 sin、cos 來編碼。選用 sin、cos 來編碼的主要原因在於 sin、cos

可以表示兩個向量之間的線性關係，並呈現不同詞語間的相堆位置，此外，sin、cos 比較不會受限於序列長度的限制。將 input embedding 與 positional encoding 相加後，會進入 multi-head attention，關於 self attention 部分，論文採用 scaled dot product attention，其公式與架構如下



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

Multi-head attention 與 self attention 差異只有在 multi-head attention 會先將 query、key、value 拆分成多個低維度向量，架構與公式如下



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

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

論文中採用 h = 8，而$d_k = d_v = \frac{d_{model}}{h} = 64$。經過 multi-head

attention 後會進入 residual connection 和 layer normalization，接著再進入 Feed forward，Feed forward 包含兩層的線性轉換層與一層 ReLU 夾在中間，forward path 為 $\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$ 。

Decoder 部分主要有加入 masked multi-head attention，實現方式與作用已經在實作部分說明。

訓練部分，optimizer 採用 Adam，參數配置為$\beta_1 = 0.9$、$\beta_2 = 0.98$、$\epsilon = 10^{-9}$。learning rate 部分，採用下列公式在訓練過程中調整 learning rate 大小

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

Normalization 部分，在與 sub-layer 相加和 normalization 之前，以及在 positional encoding 與 input embedding 相加時，都有使用 dropout。此外在訓練過程中也有使用 label smoothing。

- Layer Normalization :

  在 transformer 架構中所使用的 normalization 為 layer normalization。Layer normalization 會獨立對每層的所有特徵進行正規化，代表說每層的 activation 的 mean 和 variance 會被獨立計算，然後 activation output 會被縮放和平移，以達到 normal distribution。Layer normalization 的公式如下

  ```
  y_i = γ * (x_i - μ) / √(σ^2 + ε) + β
  ```

  1. 優點 :
     - 減少內部 covariance 偏移的影響，提高訓練的穩定性。
     - 緩解梯度消失的問題。
     - 使網路對權重與 bias 的初始值不那麼敏感。
  2. 缺點 :
     - 增加計算成本。
     - Layer normalization 中兩個可學習的超參數 scale 和 shift 對網路表現有很大的影響。
     - 在某些情況下，layer normalization 會放大數劇中的 noise。
     - 與 batch normalization 相比，靈活性比較低。
  3. 與 batch normalization 比較 :
     - Layer normalization 是獨立對每層的所有特徵進行正規化，而

batch normalization 則是對 mini-batch 正規化。

- Layer normalization 需要的記憶體空間較少，因為他不需要對整個 mini-batch 儲存 mean 和 variance。
- 對於 RNN 或是 transformer 來說，layer normalization 會有比較好的表現，因為 layer normalization 不會受 batch size 的影響。

六、 Reference

[1] Google " Attention Is All You Need" https://arxiv.org/pdf/1706.03762.pdf

[2] Sujatha Mudadla "Layer Normalization" https://medium.com/@sujathamudadla1213/layer-normalization-48ee115a14a4

[3] Hunter Phillips "Positional Encoding" https://medium.com/@hunter-j-phillips/positional-encoding-7a93db4109e6

[4] Minhajul Hoque "A comprehensive overview of transformer-based models : encoders, decoders, and more" https://medium.com/@minh.hoque/a-comprehensive-overview-of-transformer-based-models-encoders-decoders-and-more-e9bc0644a4e5

[5] OpenAI. (2023). ChatGPT (Mar 14 version) [Large language model]. https://chat.openai.com/