深度學習 - 第十一章

Status	Book
Created time	@January 22, 2025 5:13 PM

第十一章

Google Colab 檔案

Ch11.ipynb

11-1 概述自然語言處理（natural language processing, NLP）

- 電腦科學領域中，稱人類的語言（如英文或中文）為自然語言（natural language），以區隔機器語言（如 Assembly（組合語言）、LISP、XML）。
- 現代 NLP：使用機器學習和大規模資料集，讓電腦有能力不去理解語言所描述的內容，而是直接將一段語言作為輸入，然後回傳一些有用的東西
  - 「這段文字的主題是什麼？」（文字分類，text classification）。
  - 「這段文字是否有暴力的內容？」（內容過濾，content filtering）。
  - 「這段文字是正面還是負面？」（情感分析，sentiment analysis）。
  - 「在這個不完整的句子中，下一個單字會是什麼？」（語言模型，language modeling）。
- 訓練的文字處理模型不會理解語言，它們只是在輸入資料中尋找統計規則性。
- 如今大多數的 NLP 系統都基於 Transformer 架構。

11-2 準備文字資料

- 深度學習模型是可微分的函數，不能以原始文字作為輸入。
- 單字向量化（vectorizing），將文字轉換成數值張量的過程，有許多種做法，但都遵循相同模板
  1. 文字標準化（standardize），使其更易於處理。EX：轉換成小寫字母，或刪除標點符號等。
  2. 把文字分割成一個個單元（稱之為 token），這個動作叫作斷詞（tokenization）。EX：字元、單字或單字組。
  3. 把 token 轉換成一個數值向量，通常會先為資料中所有 token 加上索引。EX：one-hot 向量。

11-2-1 文字標準化（text standardization）

- 原始例子
  - 「sunset came. i was staring at the Mexico sky. Isnt nature splendid??」。
  - 「Sunset came. I stared at the México sky. Isn’t nature splendid?」。
  - 文字標準化是特徵工程的一種基礎形式，目的是消除不希望模型去處理的編碼差異。EX：「i」和「I」是同一字元、「staring」和「stared」是同一詞的不同形式。
- 標準化例子（將文字轉換成小寫字母並刪除標點符號）
  - 「sunset came i was staring at the mexico sky isnt nature splendid」。
  - 「sunset came i stared at the méxico sky isnt nature splendid」。
  - 兩個句子更相似了。
- 另一種常見轉換，將特殊字元轉換成標準字母。EX：「méxico」→「mexico」。
- 較少使用的進階表準化作法，字根提取（stemming），將詞彙的不同變形轉換成單一的通用表示法。EX：「caught」和「been catching」→「catch」。
- 透過標準化，模型不需要那麼多訓練資料，同時更能普適化。
  - 也可能抹去一些資訊，要特別注意語境。EX：「從採訪文章中萃取問題」的模型，「?」應該要被當成一個單獨的 token，而非刪除。

11-2-2 拆分文字（斷詞，tokenization）

- 標準化後，把它分解成要被向量化的單元（即 token），這步驟稱為斷詞（tokenization）。
  - 單字層級的 tokenization（word-level tokenization）：token 間以空格（或標點符號）分隔的子字串。
    - 該作法其中一種變形，在適當情況下進一步將單字拆分成「子字（subword）」。EX：「staring」→「star + ing」、「called」「call + ed」。
  - N-gram tokenization：token 是由 N 個連續單字構成的組合。
    - EX：「the cat」「he was」都是 2-gram 的 token（也稱為 bigram）。
  - 字元層級的 tokenization（character-level tokenization）：每個字元就是一個 token。
    - 在實際案例中很少使用，只有在特定情境（Ex：文字生成或語音辨識）才會看到。
- 文字處理模型
  - 「注重單字順序」的模型，稱為序列模型（sequence model）。
    - 跟序列式模型（sequential model）完全不相同，序列式模型是 Keras 建構模型的一種方式。
    - 使用單字層級的 tokenization。
  - 「將輸入單字視為一個集合、拋棄原始順序」的模型，稱為詞袋模型（bag-of-words model）。
    - 使用 N-gram tokenization，用人工方式，將少量局部的單字順序資訊注入模型。

11-2-3 建立索引

```
vocabulary = {}
   for text in dataset:
      text = standardize(text) # 將文字表準化
      tokens = tokenize(text) # 將文字分解成 token
      for token in tokens:
```

```
        # 若當前 token 不在 vocabulary 中，添加一個新項目
        # key 為當前的 token，value 為當前 vocabulary 的長度
        if token not in vocabulary:
            vocabulary[token] = len(vocabulary)

# 轉換成 one-hot 向量
def one_hot_encode_token(token):
    vector = np.zeros((len(vocabulary),)) # 向量的長度為 vocabulary 的長度
    token_index = vocabulary[token] # 取得特定 token 對應到的整數
    vector[token_index] = 1 # 將該整數對應到的位置值設為 1
    return vector
```

- 通常只會將訓練資料中，前 20,000 或 30,000 個最常見的單字放進詞彙表中。
    - 為罕見單字建立索引，會導致特徵空間過大，且這些單字通常不具備有用的資訊。
- 當詞彙表中查詢 token 時，它並不一定存在，可能會導致 KeyError。
    - 處理這種問題，應該增加一個索引來代表「不在詞彙表中的 token」（out of vocabulary index, 簡稱 OOV index）。
        - 通常使用索引 1，用 “[UNK]” 來表示對應的 token（OOV token）。
        - 代表「這裡有一個無法辨認的單字」。
    - 遮罩（mask）token
        - 使用索引 0。
        - 代表「請忽略我！我不是一個單字」。
        - 用來填補序列資料，因為在批次資料中，所有序列的程度都要相等才行，較短的序列要被填補成最長序列的長度。

11-2-4 使用 TextVectorization 層

- 單純 Python 實作

```
test_sentence = "I write, rewrite, and still rewrite again"
# 句子 -> 整數序列
encoded_sentence = vectorizer.encode(test_sentence)
print(encoded_sentence)

# 整數序列 -> 文字
decode_sentence = vectorizer.decode(encoded_sentence)
print(decode_sentence)
```

```
[2, 3, 5, 7, 1, 5, 6]
i write rewrite and [UNK] rewrite again
```

- 效能不高。

- **使用 Keras 的 TextVectorization 層**

```
[27] # 編碼
vocabulary = text_vectorization.get_vocabulary()
test_sentence = "I write, rewrite, and still rewrite again"
encoded_sentence = text_vectorization(test_sentence)
print(encoded_sentence)

# 解碼
inverse_vocab = dict(enumerate(vocabulary))
decode_sentence = " ".join(inverse_vocab[int(i)] for i in encoded_sentence)
print(decode_sentence)
```

```
tf.Tensor([ 7  3  5  9  1  5 10], shape=(7,), dtype=int64)
i write rewrite and [UNK] rewrite again
```

- 高速且高效。
- 可以直接放在 tf.data 工作流或 Keras 模型中使用

- 在 tf.data 工作流中使用 TextVectorization 層，或將其作為模型的一部份
  - TextVectorization 層主要是查找字典的操作，不能在 GPU（或 TPU） 上執行，只能在 CPU 上執行。
  - 效率高（放在 tf.data 工作流中）

```
# string_dataset 會生成字串張量
int_sequence_dataset = string_dataset.map(
```

```
        text_vectorization,
        num_parallel_calls = 4 # 在多個 CPU 核心上平行化 map() 呼叫
)
```

- 效率較差（變成模型的一部份）

```
# 接收字串的 Input 物件
text_input = keras.Input(shape=(), dtype="string")
# 將 Input 物件輸入神經層
vectorized_text = text_vectorization(text_input)
# 不斷接上新的層
embedded_input = keras.layers.Embedding(...)(vectorized_text)
output = ...
model = keras.Model(text_input, output)
```

11-3 表示單字組的兩種方法：集合（set）及序列（sequence）

- 表示單字順序是一個關鍵問題。
  - 無序的單字集合，使用詞袋模型（bag-of-words model）。
  - RNN 與 Transformer 都會考慮到順序，稱為序列模型（sequence model）

11-3-1 準備 IMDB 影評資料 → 下載並劃分訓練集與驗證集（請參考 colab 實作）

11-3-2 將單字視為一組集合：詞袋法（bag-of-words）

- 最簡單的編碼方式就是拋棄順序，將其視為一組（或一「袋」）的 token。
- 可以檢視個別單字（unigram），或嘗試透過檢視連續的 token（N-gram）來恢復一些局部的順序資訊。
- 二元編碼（binary encoding）的單一單字（unigram）
  - 使用 unigram 組成詞袋，「the cat sat on the mat」→ {"cat", "mat", "on", "sat", "the"}。

```
Model: "functional"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 20000)	0
dense (Dense)	(None, 16)	320,016
dropout (Dropout)	(None, 16)	0
dense_1 (Dense)	(None, 1)	17

```
Total params: 320,033 (1.22 MB)
Trainable params: 320,033 (1.22 MB)
Non-trainable params: 0 (0.00 B)
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 11s 16ms/step - accuracy: 0.7794 - loss: 0.4864 - val_accuracy: 0.8844 - val_loss: 0.2936
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 6s 10ms/step - accuracy: 0.8992 - loss: 0.2664 - val_accuracy: 0.8846 - val_loss: 0.3004
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - accuracy: 0.9162 - loss: 0.2305 - val_accuracy: 0.8848 - val_loss: 0.3254
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - accuracy: 0.9276 - loss: 0.2163 - val_accuracy: 0.8862 - val_loss: 0.3353
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 10s 15ms/step - accuracy: 0.9331 - loss: 0.2071 - val_accuracy: 0.8846 - val_loss: 0.3591
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - accuracy: 0.9397 - loss: 0.1933 - val_accuracy: 0.8792 - val_loss: 0.3815
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 13s 13ms/step - accuracy: 0.9424 - loss: 0.1928 - val_accuracy: 0.8830 - val_loss: 0.3794
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 10s 12ms/step - accuracy: 0.9407 - loss: 0.1948 - val_accuracy: 0.8790 - val_loss: 0.3927
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - accuracy: 0.9436 - loss: 0.1850 - val_accuracy: 0.8806 - val_loss: 0.4073
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 9s 10ms/step - accuracy: 0.9475 - loss: 0.1808 - val_accuracy: 0.8746 - val_loss: 0.4221
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 7ms/step - accuracy: 0.8870 - loss: 0.2828
Test acc: 0.883
```

    - 這個案例中，資料集是平衡的（正面與負面樣本數量相等），基準線大概會落在 50%。

- 二元編碼的 bigram
    - 透過檢視 N-gram（最常見的是 bigram）而非單字，將局部順序資訊重新注入詞袋表示法中。
    - 變成，{"the", "the cat", "cat", "cat sat", "sat", "sat on", "on", "on the", "the mat", "mat"}。

```
Model: "functional_1"
```

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 20000)	0
dense_2 (Dense)	(None, 16)	320,016
dropout_1 (Dropout)	(None, 16)	0
dense_3 (Dense)	(None, 1)	17

```
 Total params: 320,033 (1.22 MB)
 Trainable params: 320,033 (1.22 MB)
 Non-trainable params: 0 (0.00 B)
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 12s 18ms/step - accuracy: 0.7864 - loss: 0.4651 - val_accuracy: 0.8958 - val_loss: 0.2746
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 6s 9ms/step - accuracy: 0.9094 - loss: 0.2466 - val_accuracy: 0.8980 - val_loss: 0.2878
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 6s 10ms/step - accuracy: 0.9345 - loss: 0.2030 - val_accuracy: 0.8972 - val_loss: 0.3100
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - accuracy: 0.9409 - loss: 0.1814 - val_accuracy: 0.8936 - val_loss: 0.3447
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9481 - loss: 0.1749 - val_accuracy: 0.8968 - val_loss: 0.3393
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - accuracy: 0.9536 - loss: 0.1588 - val_accuracy: 0.8950 - val_loss: 0.3620
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - accuracy: 0.9570 - loss: 0.1623 - val_accuracy: 0.8986 - val_loss: 0.3693
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 6s 10ms/step - accuracy: 0.9563 - loss: 0.1698 - val_accuracy: 0.8942 - val_loss: 0.3761
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - accuracy: 0.9607 - loss: 0.1579 - val_accuracy: 0.8982 - val_loss: 0.3919
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 6s 9ms/step - accuracy: 0.9597 - loss: 0.1468 - val_accuracy: 0.8948 - val_loss: 0.3915
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 7ms/step - accuracy: 0.8969 - loss: 0.2618
Test acc: 0.897
```

    - 準確度提升（88.3% → 89.7%），局部順序是相當重要的，

- 使用 TF-IDF 編碼的 bigram
    - 計算每個單字或 N-gram 的出現次數，增添一些資訊。
    - 變成，{"the": 2, "the cat": 1, "cat": 1, "cat sat": 1, "sat": 1, "sat on": 1, "on": 1, "on the": 1, "the mat": 1, "mat": 1}。

```
text_vectorization = TextVectorization(
    ngrams=2,
    max_tokens=20000,
    output_mode="count", # 傳回 bigram 的出現次數
)

# 無法解決某些出現更頻繁的單字。EX：「the」、「a」、「is」和「are」等。
# 用 TF-IDF 正規化解決
```

- TF-IDF 正規化（Term Frequency, Inverse Document Frequency normalization）
  - 將「詞彙頻率（term frequency）」（在目前文件中的出現次數）除以「文件頻率（document frequency）」（在整個資料集中出現的頻率），對這個詞彙進行加權。

```
# 計算某個詞彙（term）在某個資料集（dataset） 的某文件（document） 中
def tfidf(term, document, dataset):
    # 計算詞彙頻率
    term_freq = document.count(term)
    # 計算文件頻率（對加總結果取自然對數）
    doc_freq = math.log(sum(doc.count(term) for doc in dataset) + 1)
    return term_freq / doc_freq
```

```
Model: "functional_3"
```

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 20000)	0
dense_6 (Dense)	(None, 16)	320,016
dropout_3 (Dropout)	(None, 16)	0
dense_7 (Dense)	(None, 1)	17

```
 Total params: 320,033 (1.22 MB)
 Trainable params: 320,033 (1.22 MB)
 Non-trainable params: 0 (0.00 B)
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 14s 20ms/step - accuracy: 0.7224 - loss: 0.7059 - val_accuracy: 0.8904 - val_loss: 0.2972
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - accuracy: 0.8718 - loss: 0.3258 - val_accuracy: 0.8832 - val_loss: 0.3036
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - accuracy: 0.8853 - loss: 0.2938 - val_accuracy: 0.8880 - val_loss: 0.3101
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 6s 9ms/step - accuracy: 0.9015 - loss: 0.2556 - val_accuracy: 0.8928 - val_loss: 0.3222
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - accuracy: 0.9016 - loss: 0.2538 - val_accuracy: 0.8812 - val_loss: 0.3394
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - accuracy: 0.9066 - loss: 0.2287 - val_accuracy: 0.8760 - val_loss: 0.3526
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - accuracy: 0.9123 - loss: 0.2210 - val_accuracy: 0.8850 - val_loss: 0.3759
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 10s 8ms/step - accuracy: 0.9135 - loss: 0.2219 - val_accuracy: 0.8842 - val_loss: 0.3746
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - accuracy: 0.9157 - loss: 0.2164 - val_accuracy: 0.8828 - val_loss: 0.3919
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - accuracy: 0.9142 - loss: 0.2191 - val_accuracy: 0.8764 - val_loss: 0.3909
782/782 ━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step - accuracy: 0.8927 - loss: 0.2947
Test acc: 0.889
```

- 準確率（89.7% → 88.9）並沒有什麼提升。

    - 對於許多文字分類資料集而言，比起一般的二元編碼，使用 TF-IDF 往往能讓準確度上升 1 個百分點。

11-3-3 將單字作為序列處理：序列模型（sequence model）

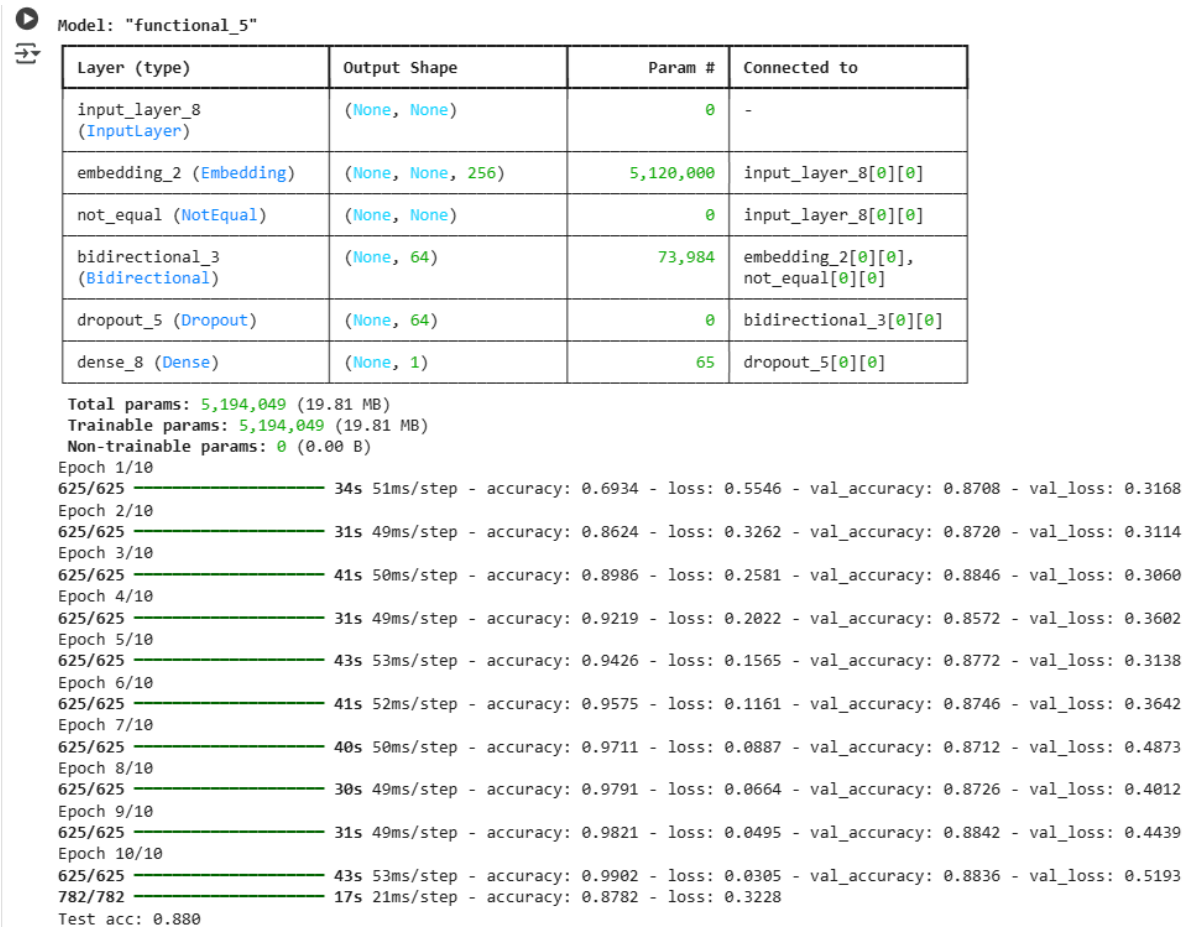
- 第一個實際案例
    - 訓練速度非常慢，因為輸入相當大，每個輸入樣本都被編碼成一個大小為（600, 20000） 的矩陣，一篇影評就有 12,000,000 個浮點數。
    - 準確度（87%）遠不及二元 unigram 模型。
- 認識詞嵌入法（word embedding）
    - 兩個單字向量間的幾何關係（geometric relationship），應該要能反映單字之間的語意關係。
    - 相似單字會被嵌入到相近的位置，嵌入空間中的特定方向是有意義的。
        - 「gender（性別）」向量和「plural（複數）」向量
            - 「king（國王）」向量加上「female（女性）」向量，得到「queen（皇后）」向量。
            - 「king（國王）」向量加上「plural（複數）」向量，得到「kings（國王們）」向量。
    - 在實際案例中使用
        - 針對欲解決任務進行詞嵌入向量的學習。
        - 將預先學習好的詞嵌入向量載入模型中使用，稱為預訓練詞嵌入法（pretrained word embedding）。
- 用 Embedding 層學習詞嵌入向量
    - 可以理解成，會把「整數索引（代表特定單字）」對應到「密集向量」的 Python 字典。
    - 隨機初始權重，訓練過程中，經由反向傳播逐步調整，將空間結構化成下游模型能利用的東西。
    - 訓練完成後，嵌入空間會展現豐富的結構性，這種結構式專門為了當前任務而存在的。

```
Model: "functional_4"
```

Layer (type)	Output Shape	Param #
input_layer_7 (InputLayer)	(None, None)	0
embedding_1 (Embedding)	(None, None, 256)	5,120,000
bidirectional_2 (Bidirectional)	(None, 64)	73,984
dropout_4 (Dropout)	(None, 64)	0
dense_7 (Dense)	(None, 1)	65

```
 Total params: 5,194,049 (19.81 MB)
 Trainable params: 5,194,049 (19.81 MB)
 Non-trainable params: 0 (0.00 B)
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 34s 49ms/step - accuracy: 0.6321 - loss: 0.6241 - val_accuracy: 0.7252 - val_loss: 0.5347
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 34s 54ms/step - accuracy: 0.8290 - loss: 0.4259 - val_accuracy: 0.8462 - val_loss: 0.3699
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 33s 53ms/step - accuracy: 0.8718 - loss: 0.3341 - val_accuracy: 0.8538 - val_loss: 0.3546
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 32s 50ms/step - accuracy: 0.8970 - loss: 0.2816 - val_accuracy: 0.8778 - val_loss: 0.3231
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 40s 49ms/step - accuracy: 0.9124 - loss: 0.2443 - val_accuracy: 0.8804 - val_loss: 0.3192
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 32s 52ms/step - accuracy: 0.9273 - loss: 0.2083 - val_accuracy: 0.8870 - val_loss: 0.3206
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 31s 49ms/step - accuracy: 0.9421 - loss: 0.1693 - val_accuracy: 0.8870 - val_loss: 0.3427
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 48ms/step - accuracy: 0.9549 - loss: 0.1379 - val_accuracy: 0.8772 - val_loss: 0.3693
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 43s 52ms/step - accuracy: 0.9644 - loss: 0.1106 - val_accuracy: 0.8748 - val_loss: 0.4281
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 40s 50ms/step - accuracy: 0.9736 - loss: 0.0831 - val_accuracy: 0.8774 - val_loss: 0.4966
782/782 ━━━━━━━━━━━━━━━━━━━━ 17s 21ms/step - accuracy: 0.8591 - loss: 0.3647
Test acc: 0.861
```

    - 速度比 one-hot 模型快很多，且測試準度不相上下，但離普通 bigram 模型還有一大段距離。
- 認識填補（padding） 和遮罩（masking）
    - 輸入序中充滿了零，因為少於 600 個 token 的句子在末尾處用零填補。
    - 告訴 RNN 應該跳過迭代（只會看到代表填補 token 的向量），使用遮罩（masking）。
        - 實際使用時，幾乎不用手動管理遮罩。



Model: "functional_5"

Layer (type)	Output Shape	Param #	Connected to
input_layer_8 (InputLayer)	(None, None)	0	-
embedding_2 (Embedding)	(None, None, 256)	5,120,000	input_layer_8[0][0]
not_equal (NotEqual)	(None, None)	0	input_layer_8[0][0]
bidirectional_3 (Bidirectional)	(None, 64)	73,984	embedding_2[0][0], not_equal[0][0]
dropout_5 (Dropout)	(None, 64)	0	bidirectional_3[0][0]
dense_8 (Dense)	(None, 1)	65	dropout_5[0][0]

```
Total params: 5,194,049 (19.81 MB)
Trainable params: 5,194,049 (19.81 MB)
Non-trainable params: 0 (0.00 B)
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 34s 51ms/step - accuracy: 0.6934 - loss: 0.5546 - val_accuracy: 0.8708 - val_loss: 0.3168
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 31s 49ms/step - accuracy: 0.8624 - loss: 0.3262 - val_accuracy: 0.8720 - val_loss: 0.3114
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 50ms/step - accuracy: 0.8986 - loss: 0.2581 - val_accuracy: 0.8846 - val_loss: 0.3060
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 31s 49ms/step - accuracy: 0.9219 - loss: 0.2022 - val_accuracy: 0.8572 - val_loss: 0.3602
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 43s 53ms/step - accuracy: 0.9426 - loss: 0.1565 - val_accuracy: 0.8772 - val_loss: 0.3138
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 52ms/step - accuracy: 0.9575 - loss: 0.1161 - val_accuracy: 0.8746 - val_loss: 0.3642
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 40s 50ms/step - accuracy: 0.9711 - loss: 0.0887 - val_accuracy: 0.8712 - val_loss: 0.4873
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 30s 49ms/step - accuracy: 0.9791 - loss: 0.0664 - val_accuracy: 0.8726 - val_loss: 0.4012
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 31s 49ms/step - accuracy: 0.9821 - loss: 0.0495 - val_accuracy: 0.8842 - val_loss: 0.4439
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 43s 53ms/step - accuracy: 0.9902 - loss: 0.0305 - val_accuracy: 0.8836 - val_loss: 0.5193
782/782 ━━━━━━━━━━━━━━━━━━━━ 17s 21ms/step - accuracy: 0.8782 - loss: 0.3228
Test acc: 0.880
```

    - 準確度有些維進步（87% → 88%）。

- 使用預先訓練的詞嵌入向量
    - GloVe（Global Vectors for Word Representation），該嵌入法技術的基礎，是將單字共同出現的統計矩陣進行分解。

Model: "functional_6"

Layer (type)	Output Shape	Param #	Connected to
input_layer_9 (InputLayer)	(None, None)	0	-
embedding_3 (Embedding)	(None, None, 100)	2,000,000	input_layer_9[0][0]
not_equal_2 (NotEqual)	(None, None)	0	input_layer_9[0][0]
bidirectional_4 (Bidirectional)	(None, 64)	34,048	embedding_3[0][0], not_equal_2[0][0]
dropout_6 (Dropout)	(None, 64)	0	bidirectional_4[0][0]
dense_9 (Dense)	(None, 1)	65	dropout_6[0][0]

```
Total params: 2,034,113 (7.76 MB)
Trainable params: 34,113 (133.25 KB)
Non-trainable params: 2,000,000 (7.63 MB)
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 63ms/step - accuracy: 0.6359 - loss: 0.6299 - val_accuracy: 0.7748 - val_loss: 0.4777
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 41s 62ms/step - accuracy: 0.7804 - loss: 0.4797 - val_accuracy: 0.8076 - val_loss: 0.4228
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 39s 60ms/step - accuracy: 0.8100 - loss: 0.4231 - val_accuracy: 0.8360 - val_loss: 0.3762
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 35s 50ms/step - accuracy: 0.8298 - loss: 0.3938 - val_accuracy: 0.8274 - val_loss: 0.3867
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 48s 62ms/step - accuracy: 0.8430 - loss: 0.3632 - val_accuracy: 0.8318 - val_loss: 0.3702
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 39s 60ms/step - accuracy: 0.8518 - loss: 0.3386 - val_accuracy: 0.8480 - val_loss: 0.3439
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 43s 62ms/step - accuracy: 0.8644 - loss: 0.3186 - val_accuracy: 0.8660 - val_loss: 0.3152
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 42s 64ms/step - accuracy: 0.8761 - loss: 0.2980 - val_accuracy: 0.8710 - val_loss: 0.3100
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 36s 56ms/step - accuracy: 0.8820 - loss: 0.2844 - val_accuracy: 0.8452 - val_loss: 0.3612
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 39s 63ms/step - accuracy: 0.8884 - loss: 0.2703 - val_accuracy: 0.8806 - val_loss: 0.3000
782/782 ━━━━━━━━━━━━━━━━━━━━ 21s 26ms/step - accuracy: 0.8730 - loss: 0.3061
Test acc: 0.878
```

    - 模型準確度僅達到 87.8%，因此在這個特定任務中，預訓練的嵌入向量並不是很有用。
        - 因為現有資料集中已包含足夠樣本。

11-4 Transformer 架構

11-4-1 認識 self-attention 機制

- 模型應該對某些特徵「更加關注（pay more attention）」，而對其他特徵「少關注一點（pay less attention）」。
- 兩次類似概念
    - 卷積神經網路中的最大池化（max pooling）操作，只著眼於某個空間區域中的特徵池，並選擇一個特徵保留。

    - TF-IDF 正規化，根據不同 token 可能乘載的資訊量多寡，為 token 的重要性進行評分（加權）。
- 這種 attention 機制可以用來凸顯或抹去某些特徵，還能讓特徵具有語境感知能力（context-aware）。
- 依其語境（周圍單字）提供不同的向量表示法。
- EX：「The train left the station on time」，station 指的是哪一種 station。
    1. 計算「station」與句子中其他單字間的關聯分數，也就是「attention 分數」。
        - 計算兩個單字向量間的點積（dot product），以衡量它們間的關聯強度。
    2. 計算句子中所有單字向量的加權總合（以 attention 分數進行加權）。
        - 產生向量會是「station」結合周圍語境的新表示法。
    - Keras 有內建的層來進行處理，即 MultiHeadAttention 層

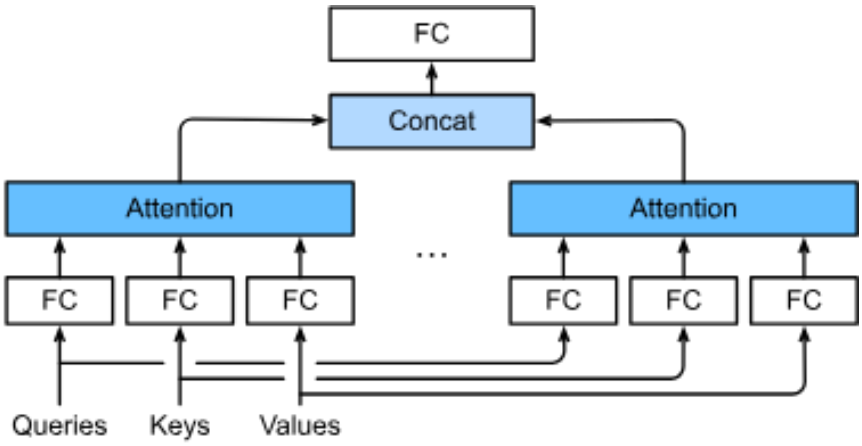
```
num_heads = 4
embed_dim = 256
mha_layer = MultiHeadAttrntion(num_heads=num_heads, key_dim=embed_d
outputs = mha_layer(inputs, inputs, inputs)
```

- 普世化的 self-attention：query-key-value 模型
    - Transformer 架構是一種 Seq2seq（sequence-to-sequence，序列至序列）模型。
    - 針對 query 中每個元素，計算該元素與每個 key 元素的關聯性強度，並使用這些分數對 value 元素進行加權總合。
        - EX：輸入 query，要從資料庫中取得一張照片，「海灘上的狗」。資料庫內部，每張圖都會由一組關鍵字所描述，如「貓」、「狗」、「派對」等等，稱為「key」。
            1. 搜尋引擎把 query 與資料庫中的 key 進行比較。
            2. 搜尋引擎依照匹配度（關聯性）排序這些 key，並傳回前 N 個匹配度最高的圖片。
    - 如果只是做序列分類，那麼 query、key 跟 value 都是一樣的。

    - 將一個序列和自己做比較，用整個序列的語境來豐富每個 token 的資訊。
    - 這就是為何要傳遞 3 次 inputs 到 MultiHeadAttention 層。

11-4-2 多端口 attention（Multi-head attention）

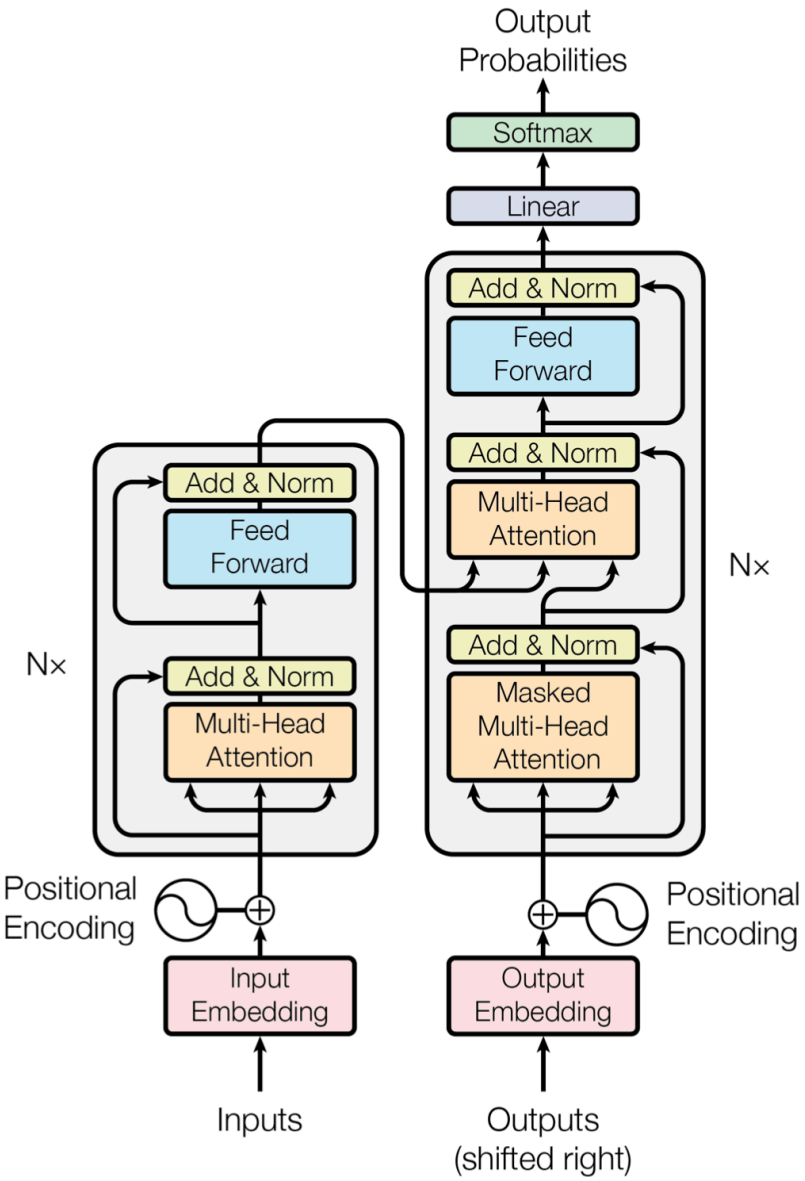
- 「多端口」代表 self-attention 層的輸出空間會被分解成多個獨立的子空間，且是個別學習而來。
    1. 初始的 query、key 和 value 會經過 3 組獨立的密集投影（Dense 層運算）。
    2. 透過 neural attention 進行處理，形成單一的輸出序列。
    3. 每個子空間輸出序列串接，形成最後的輸出。
    - 每個這樣的子空間稱為「端口（head）」。



- 獨立的 head 有助於為每個 token 學習不同的特徵組合。

11-4-3 Transformer 編碼器

- 將 MultiHeadAttention 層跟密集投影（Dense 層）串接，並添加正規化和殘差連接。



- Transformer 架構由兩部分組成
  - 處理原始序列的 Transformer 編碼器。
    - 也可用於文字分類，是個很通用的模組，可以擷取一個序列，並學習將它變成更有用的表示法。
  - 透過原始序列來生成翻譯結果的 Transformer 解碼器。
  - 在影評分類任務上實作 Transformer 編碼器

```
...

class TransformerEncoder(layers.Layer):
  def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
    super().__init__(**kwargs)
    self.embed_dim = embed_dim # 輸入 token 的向量大小
    self.dense_dim = dense_dim # 內部密集層的大小
    self.num_heads = num_heads # attention 端口的數量
    self.attention = layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=embed_dim)
    self.dense_proj = keras.Sequential(
        [layers.Dense(dense_dim, activation="relu"),
         layers.Dense(embed_dim),]
    )
    # 使用 LayerNormalization 層
    self.layernorm_1 = layers.LayerNormalization()
    self.layernorm_2 = layers.LayerNormalization()

...
```

- BatchNormalization 層在序列資料上表現不好，改用 LayerNormalization 層，它會對批次中的每個序列分別進行正規化處理。
  - BatchNormalization 層會收集很多樣本的資訊，以獲取準確的特徵平均值和變異數統計結果。
  - LayerNormalization 層分別針對每個序列中的資料進行池化。

```
Epoch 1/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 69s 95ms/step - accuracy: 0.5501 - loss: 0.8613 - val_accuracy: 0.8282 - val_loss: 0.3896
Epoch 2/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 57s 92ms/step - accuracy: 0.8128 - loss: 0.4141 - val_accuracy: 0.8192 - val_loss: 0.3947
Epoch 3/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 84s 94ms/step - accuracy: 0.8447 - loss: 0.3556 - val_accuracy: 0.8602 - val_loss: 0.3252
Epoch 4/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 59s 95ms/step - accuracy: 0.8579 - loss: 0.3335 - val_accuracy: 0.8692 - val_loss: 0.3109
Epoch 5/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 85s 100ms/step - accuracy: 0.8705 - loss: 0.3048 - val_accuracy: 0.8722 - val_loss: 0.2977
Epoch 6/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 62s 100ms/step - accuracy: 0.8806 - loss: 0.2824 - val_accuracy: 0.8772 - val_loss: 0.2930
Epoch 7/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 79s 96ms/step - accuracy: 0.8944 - loss: 0.2659 - val_accuracy: 0.8764 - val_loss: 0.3011
Epoch 8/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 60s 96ms/step - accuracy: 0.8969 - loss: 0.2466 - val_accuracy: 0.8786 - val_loss: 0.2894
Epoch 9/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 85s 101ms/step - accuracy: 0.9107 - loss: 0.2248 - val_accuracy: 0.8762 - val_loss: 0.2973
Epoch 10/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 79s 97ms/step - accuracy: 0.9181 - loss: 0.2072 - val_accuracy: 0.8744 - val_loss: 0.3100
Epoch 11/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 60s 96ms/step - accuracy: 0.9254 - loss: 0.1888 - val_accuracy: 0.8700 - val_loss: 0.3296
Epoch 12/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 60s 96ms/step - accuracy: 0.9321 - loss: 0.1700 - val_accuracy: 0.8752 - val_loss: 0.3201
Epoch 13/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 60s 96ms/step - accuracy: 0.9442 - loss: 0.1462 - val_accuracy: 0.8728 - val_loss: 0.3487
Epoch 14/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 82s 95ms/step - accuracy: 0.9523 - loss: 0.1299 - val_accuracy: 0.8606 - val_loss: 0.3996
Epoch 15/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 83s 97ms/step - accuracy: 0.9587 - loss: 0.1138 - val_accuracy: 0.8700 - val_loss: 0.3613
Epoch 16/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 57s 92ms/step - accuracy: 0.9656 - loss: 0.0973 - val_accuracy: 0.8666 - val_loss: 0.3786
Epoch 17/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 81s 90ms/step - accuracy: 0.9691 - loss: 0.0853 - val_accuracy: 0.8634 - val_loss: 0.4151
Epoch 18/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 81s 89ms/step - accuracy: 0.9719 - loss: 0.0743 - val_accuracy: 0.8672 - val_loss: 0.4348
Epoch 19/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 80s 86ms/step - accuracy: 0.9760 - loss: 0.0694 - val_accuracy: 0.8452 - val_loss: 0.4650
Epoch 20/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 83s 88ms/step - accuracy: 0.9767 - loss: 0.0625 - val_accuracy: 0.8590 - val_loss: 0.4795
/usr/local/lib/python3.11/dist-packages/keras/src/layers/layer.py:393: UserWarning: `build()` was called on layer 'transformer_encoder', how
  warnings.warn(
782/782 ━━━━━━━━━━━━━━━━━━━━ 14s 17ms/step - accuracy: 0.8714 - loss: 0.3027
Test acc: 0.873
```

    - 準確度達到 87.3%，比 GRU 模型略差一點。
        - 剛剛實作的 Transformer 編碼器根本就不是序列模型。就算改變序列中 token 的順序，還是會得到完全相同的 attention 分數。
        - self-attention 是一種集合處理機制（set-processing mechanism），主要關注序列元素組合之間的關係，對元素位置毫無興趣。
- 使用位置編碼（positional encoding）重新注入順序資訊
    - 把位置嵌入向量添加到相應的詞嵌入向量中，得到一個能感知位置的詞嵌入向量，這種技術稱為位置嵌入法（positional embedding）。
- 組合所有元件：用作文字分類的 Transformer
    - 考慮單字順序，把 Embedding 層換成可感知位置的 PositionalEmbedding 層。
    - 準確度達到 88.3%，明顯進步。
        - 證明單字順序資訊在文字分類中的重要性。

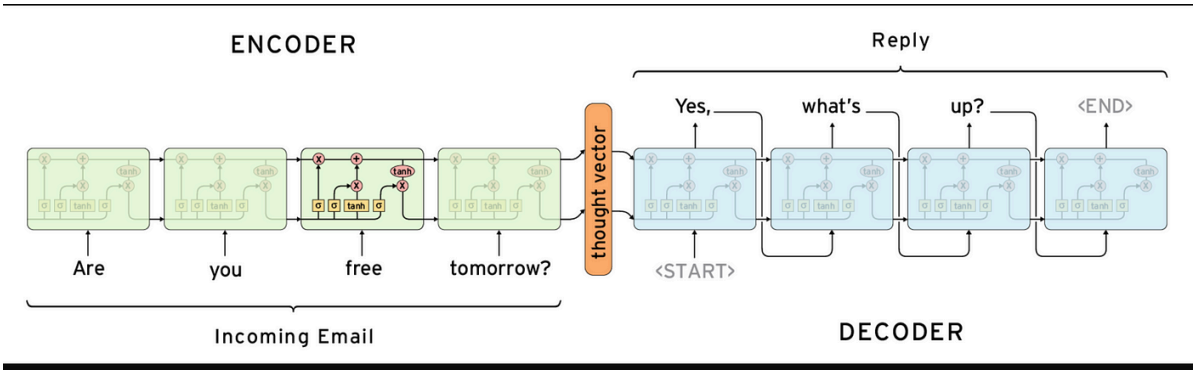
11-4-4 何時該選擇序列模型，而非詞袋模型

- 文字分類任務，應該密切注意「訓練資料中的樣本數」與「每個樣本的平均字數」之間的比率。
    - 比率小於 1,500，bigram 詞袋表現更好（訓練跟迭代速度也會更快）。
    - 比率大於 1,500，選擇序列模型。
    - 手上有大量樣本，每個樣本相對較短，序列模型會有很好的效果。
- 序列模型的輸入，代表更豐富、更複雜的空間，因此需要很多資料來映射出這個空間。
- 樣本越短，模型就越不能丟棄它包含的任何資訊。因為單字順序變得更重要了，丟棄會讓語意模糊不清。

11-5 文字分類以外的任務 - 以 Seq2seq 模型為例例

- Seq2seq 模型會把一個序列作為輸入（通常是一個句子或段落），並將它轉換成不同的序列。
    - 機器翻譯（machine translation）：原始語言的一個段落 → 目標語言的相應內容。
    - 文章摘要（text summarization）：長文件 → 保留重要資訊的簡短版本。
    - 問答（question answering）：輸入問題 → 答案。
    - 聊天機器人（chatbot）：對話提示（dialogue prompt） → 對該提示的回覆。
    - 文字生成（text generation）：文字提示 → 符合該提示的一篇文章或段落。
- Seq2seq 模型背後的普遍模板
    - 編碼器模型會把原始序列轉換成一個中間表示法的編碼向量。
    - 解碼器會學習如何透過編碼向量，以及目標序列中的先前 token（第 0 個到第 i - 1 個 token），來預測下一個（第 i 個）位置的目標 token。
- 在推論（inference）過程中，無法取得目標序列，要從頭開始逐一預測目標序列中的 token
    1. 將原始序列送入編碼器以獲取編碼向量。
    2. 解碼器會先檢視編碼向量，以及初始的「種子」token，並用來預測目標序列中的第一個 token。

3. 預測出的 token 會被送回解碼器中，進而生成下一個預測的 token，直到生成一個停止 token 為止



11-5-1 機器翻譯的案例 → 下載英文 - 西班牙文翻譯資料集，進行資料處理（請參考 colab 實作）

- 解析原始檔案
  - 在西文內容的前方加入「[start]」字元，後方加入「[end]」字元。
- 資料洗牌並分成訓練集（70%）、驗證集（15%）、測試集（15%）。
- 準備兩個獨立的 TextVectorization 層，分別用於英文與西文。
- 自訂字串的預處理方式
  - 保留已經插入的「[start]」和「[end]」token。
  - 處理標點符號，除掉西文的「¿」字元。
- 為簡單起見，在課本案例中會去除標點符號，但在較正式的翻譯模型中，會把標點符號字元當成獨立 token ，而非直接去除。

11-5-2 用 RNN 進行 Seq2seq 的學習

- 要用 RNN 將一個序列轉換成另一個序列，最簡單的方法就是保留 RNN 每個時步的輸出（即將 return_sequences 設為 True）。但要注意兩個問題
  - 目標序列與原始序列的長度必須相同。可以進行填補，讓長度一致。
  - 由於 RNN 的「逐一處理時步」特性，使用原始序列中第 0~N 個 token，預測目標序列第 N 個 token。

        - 使用過去的資訊來預測未來。
        - 導致 RNN 不是用於大多數任務，特別是翻譯任務。
        - 當資訊量非常少時，根本不可能預測成功。
- 準確度 64%（自己跑 colab 需要跑很久，這裡直接抓課本上的數據來說）。
- 處理現實世界中的祭器翻譯系統時，可能會用「BLEU（BiLingual Evaluation Understudy）分數」來評估模型。
    - 一個能檢查整個生成序列的指標，似乎與人類對翻譯品質的感知很相似。
- 用 RNN 來進行 Seq2seq 學習還有一些先天上的限制
    - 原始序列的表示法必須完全保存在「編碼器的狀態向量」中，大大限制能夠翻譯的句子長度和複雜性。
    - RNN 無法保留長期語境，在處理非常長的序列時會遇到困難，因為 RNN 會逐漸「忘記」過去的資訊。

11-5-3 用 Transformer 進行 Seq2seq 學習

- neural attention 讓 Transformer 能比 RNN 處理更長、更複雜的序列。
- Transformer 編碼器會把編碼後的表示法以序列的方式呈現，一個具語境感知能力的嵌入向量序列。
- 因果填補（causal padding）對成功訓練 Seq2seq Transformer 是個重要關鍵。
    - TransformerDecoder 不分順序，會一次檢視整個目標序列，若使用完整的輸入，會直接學習將輸入的目標序列中的時步 N+1，直接複製到輸出的目標序列中的時步 N。
        - 模型取得完美的訓練準確度，但在推論期間是完全派不上用場的，因為超過 N 的輸入時步在推論期間是無法取得的。
    - 解決方法，把 attention 矩陣的部分資訊遮起來，移除模型對未來資訊的任何關注。
        - 在生成第 N+1 個目標 token 時，只使用目標序列中第 0~N 個 token 資訊。
        - 在 TransformerDecoder 中添加一個 get_causal_attention_mask(self, inputs)，以取得傳遞到 MultiHeadAttention 層的 attention 遮罩。
- 結合所有部件：用於機器翻譯的 Transformer

    - 訓練一個端到端 Transformer 模型，它會將「原始序列及目標序列」映射到「偏離了一個時步的目標序列」（target sequence one step in the feature）。
    - 結合 PositionalEmbedding 層、TransformerEncoder 和 TransformerDecoder。
- 準確度 67%，比使用 GRU 的模型高出許多。（自己跑 colab 需要跑很久，這裡直接抓課本上的數據來說）。
- 主觀上，Transformer 似乎比 GRU 翻譯模型表現得更好。雖然它依舊是個玩具模型。