Applied Deep Learning - Homework 2

資工碩一 R10922005 李澤諺

April 26, 2022

Problem 1-1

根據Hugging Face官方文件的說明，Hugging Face 所提供的 tokenizer 有分爲以下幾種：

Byte-Pair Encoding (BPE)

BPE 的訓練過程爲，其首先會將 dataset 之中的所有句子進行normalization 和 pre-tokenization (normalization 會進行像是移除多餘空格、將英文字母全部轉爲小寫等流程，而 pre-tokenization 可能爲以空格和標點符號進行斷詞等方式)，並將 pre-tokenization 之後的 dataset 之中有出現的所有 symbol 集合起來形成一個 base vocabulary，接著，不斷找出當前的 vocabulary 之中哪兩個 token 合併之後所形成的新的 token 會在原本的 dataset 之中出現的頻率最高，找出後便將這兩個 token 合併起來形成一個新的 token 加入當前的 vocabulary 之中，並記錄該 merge rule，如此不斷重複以上過程，直到 vocabulary 的大小到達指定的上限爲止。

而 BPE 在進行 tokenization 時，一樣會先將句子進行 normalization 和 pre-tokenization，並將 pre-tokenization 之後的句子再進一步拆分成 symbol，接著會不斷使用訓練過程所記錄的 merge rule 將當前句子中的 token 合併，直到沒有 merge rule 可以使用爲止，以此得到最終的 tokenization 結果。

如此設計的目的在於，若是將整個 dataset 之中的所有 word 都加入 vocabulary 之中，便會使得 vocabulary 的大小過大，而若是只將 character 加入 vocabulary 之中，並將所有 word 都 tokenize 成 character，雖然可以有效減少 vocabulary 的大小，且不會產生 OOV 的問題，但僅靠單一個 character 沒有辦法表示出不同詞彙所代表的複雜意義，因此 Hugging Face 中的 tokenizer 都是使用 subword 構成 vocabulary 並以此進行 tokenization，以減少前述的問題。而在 BPE 的訓練過程中，若一個 token pair 在 dataset 之中出現的頻率越高，代表其越有可能是具有意義的 subword，因此 BPE 才會選擇將出現頻率最高的 token pair 合併起來將其加入 vocabulary 之中。

但是，若 pre-tokenization 之後所產生的 base vocabulary 大小本身就已經很大 (例如使用 Unicode 時)，則訓練完後所產生的 vocabulary 大小依舊會過大，爲了解決這個問題，有些 model 會使用 byte-level BPE，意即在 pre-tokenization 之後，使用 byte 而不是 symbol 來形成 base vocabulary，如此便能有效減少 vocabulary

的大小，且不會產生 OOV 的問題。

WordPiece

WordPiece 的訓練過程和 BPE 極爲相似，兩者唯一的差別在於 token pair 的選擇方式，WordPiece 會爲每一個 token pair 計算一個分數：設當前的 vocabulary 爲 $\{x_1, x_2, \cdots, x_N\}$，而 $freq(x)$ 代表 x 在原本的 dataset 之中出現的次數，則定義 x_i 和 x_j 所形成的 token pair 之分數爲 $\frac{freq(x_i x_j)}{freq(x_i)freq(x_j)}$，WordPiece 會找出當前的 vocabulary 之中哪一個 token pair 的分數最高，並將其合併加入 vocabulary 之中，直到 vocabulary 的大小到達指定的上限爲止。

而在進行 tokenization 時，由於 WordPiece 在訓練過程中並不會紀錄 merge rule，而只會紀錄最終的 vocabulary，因此 WordPiece 在將 word 進行 tokenization 時，會不斷找出當前的 word 之中哪一個 subword 有出現在 vocabulary 之中且長度最長，WordPiece 會保留該 subword 並對該 word 的其餘部份繼續進行 tokenize，以此得到最終的 tokenization 結果。

Unigram

Unigram 的訓練過程爲，其會先產生一個極大的 base vocabulary (例如將 pre-tokenization 處理過後的 dataset 之中較常出現的 substring 集合起來作爲 base vocabulary)，接著，Unigram 會爲當前的 vocabulary 計算一個 loss (定義會在之後的段落中說明)，並計算當前的 vocabulary 之中各個 token 被移除之後會使得 loss 上升多少，並移除比例爲 p (p 通常介於 10% 20%) 移除後會使得 loss 上升較少的 token，不斷重複以上過程直到 vocabulary 大小不超過所指定的上限爲止。

而在進行 tokenization 時，對於每一個 word，Unigram 會根據 vocabulary 找出該 word 所有可能的斷詞方式，並計算每一種斷詞方式出現的機率，意即，設 word w 的其中一種可能的斷詞方式爲 $w = x_1 x_2 \cdots x_n$，而 token x_i 在 dataset 之中出現的機率爲 $P(x_i)$，則該斷詞方式出現的機率爲 $P(x_1) \times P(x_2) \times \cdots \times P(x_n)$，Unigram 會找出所有可能的斷詞方式之中出現機率最高者，並將其作爲最終的 tokenization 結果。

而在訓練過程之中，Unigram 對於 loss 的定義方式如下：Unigram 會使用當前的 vocabulary 對 dataset 之中的所有 word 進行 tokenization，令 vocabulary 記爲 V，而 word w 斷詞的結果出現的機率記爲 $P(w)$，則當前 vocabulary 的 loss 定義爲 $\sum_{w \in V} -log(P(w))$，如前所述，Unigram 會不斷根據 loss 找出要被移除的 token，直到 vocabulary 大小不超過所指定的上限爲止。

SentencePiece

SentencePiece 設計的目的在於，有些語言並不是使用空白來進行斷詞的，因此在 SentencePiece 之中，其會將句子中的空白皆視爲 character，以此將 dataset 用來訓練 BPE tokenizer 或是 Unigram tokenizer。根據 Hugging Face 的官方文件所述，其所有的 SentencePiece tokenizer 都是搭配 Unigram 訓練而得。

我於本次作業中有嘗試使用 BERT 和 MacBERT 兩種模型，兩者皆爲使用 WordPiece tokenizer，而在處理中文時，MacBERT 還會使用N-LTP來協助 tokenization。

Problem 1-2

以下爲我在訓練時所進行的 pre-processing 步驟：

(1) 首先，我會將 question 和 paragraph 進行 tokenization，設 question 和 paragraph 進行 tokenization 之後所得到的 token 分別爲 q_0、q_1、q_2、$\cdots$、q_{m-1} 和 p_0、p_1、p_2、$\cdots$、p_{n-1}。
(2) 接著，若 question 的 token 數目超過 50 個，則我會只保留 question 的前 50 個 token，意即，保留 q_0、q_1、q_2、$\cdots$、q_{k-1}，其中 $k = min(50, m)$。
(3) 而若 paragraph 的 token 數目超過 450 個，則我會先使用 tokenizer 的 char_to_token 函式，找出 answer 的第一個和最後一個 character 會分別對應到哪兩個 token，令 answer 的第一個和最後一個 character 會分別對應到 p_i 和 p_j，接著，我會從 $[0, i]$ 之間隨機選出一個整數 s，作爲要保留的 token 的起始 index，並取 $t = min(s + 449, n - 1)$，作爲要保留的 token 的結尾 index，最後，只保留 p_s、p_{s+1}、p_{s+2}、$\cdots$、p_{t-1}、p_t。
(4) 最後，我會將 CLS token 接上 q_0、q_1、q_2、$\cdots$、q_{k-1}，再接上 SEP token，再接上 p_s、p_{s+1}、p_{s+2}、$\cdots$、p_{t-1}、p_t，再接上 SEP token，再不斷接上 PAD token 直到 sequence 的長度到達 512，以此作爲 model 的 input。
(5) 綜合以上所述，可得 answer 的第一個和最後一個 character 在 model 的 input sequence 之中所對應到的 token index 分別爲 $i-s+k+2$、$j-s+k+2$。

而以下爲我在預測時所進行的 pre-processing 和 post-processing 步驟：

(1) 首先，我會將 question 和 paragraph 進行 tokenization，設 question 和 paragraph 進行 tokenization 之後所得到的 token 分別爲 q_0、q_1、q_2、$\cdots$、q_{m-1} 和 p_0、p_1、p_2、$\cdots$、p_{n-1}。
(2) 接著，若 question 的 token 數目超過 50 個，則我會只保留 question 的前 50 個 token，意即，保留 q_0、q_1、q_2、$\cdots$、q_{k-1}，其中 $k = min(50, m)$。
(3) 而若 paragraph 的 token 數目超過 450 個，則我會使用 sliding window 對 paragraph 進行切割，其中 window 的大小爲 450，而 window 之間的 stride 爲 50，因此，第 s 個 window 之中所包含的 token 爲 p_{50s}、p_{50s+1}、p_{50s+2}、$\cdots$、$p_{50s+t-2}$、$p_{50s+t-1}$，其中 $t = min(450, n - 50s)$。
(4) 最後，我會將 CLS token 接上 q_0、q_1、q_2、$\cdots$、q_{k-1}，再接上 SEP token，再接上第 s 個 window，即 p_{50s}、p_{50s+1}、p_{50s+2}、$\cdots$、$p_{50s+t-2}$、$p_{50s+t-1}$，再接上 SEP token，再不斷接上 PAD token 直到 sequence 的長度到達 512，以此作爲 model 的 input。
(5) Model 會輸出兩組 logit，將其分別記爲 $y_{0,0}$、$y_{0,1}$、$y_{0,2}$、$\cdots$、$y_{0,511}$ 和 $y_{1,0}$、$y_{1,1}$、$y_{1,2}$、$\cdots$、$y_{1,511}$，其中 $y_{0,i}$ 代表 input sequence 之中第 i 個 token 爲 answer 起始的分數，而 $y_{1,i}$ 則代表 input sequence 之中第 i 個 token 爲

answer 結尾的分數，我會找出 $i, j = argmax_{0 \leq i,j \leq 511}(y_{0,i} + y_{1,j})$，其中 i 和 j 還要滿足以下條件：

(a) $k + 2 \leq i, j \leq k + t + 1$，即 answer 必須出現在 paragraph 之中，而不能在 question 之中找 answer，且 answer 不能包含 CLS token、SEP token、PAD token。

(b) $i \leq j$，即 answer 的起始 index 不能大於結尾 index。

對於每個 window 我都會以此方式找出該 window 之中最有可能的 answer 爲何，並找出所有 window 之中哪一個 window 所找出的 answer 之起始 logit 和結尾 logit 的和最大，便將該 window 所找出的 answer 作爲整個 paragraph 之中所預測出的 answer。

(6) 設最後所找到的 answer 是在第 s 個 window 之中，且 answer 的起始 token 和結尾 token 在 model 的 input sequence 之中 index 分別爲 i 和 j，則 answer 的起始 token 與結尾 token 在 tokenize 過後的 paragraph 之中其實分別爲 $p_{50s+i-k-2}$ 和 $p_{50s+j-k-2}$，爲了避免直接使用 tokenizer 對其進行 decode 可能會產生 UNK token 的問題，我選擇使用 tokenizer 的 token_to_chars 函式，找出 $p_{50s+i-k-2}$ 和 $p_{50s+j-k-2}$ 在最原本的 paragraph 之中對應到哪些 character，並將這之間的所有 character 截取下來，作爲最終預測的結果。

Problem 2-1

以下爲我在 multiple choice 之中所使用的 BERT 的 configuration：

_name_or_path	"bert-base-chinese"
architectures	["BertForMultipleChoice"]
attention_probs_dropout_prob	0.1
classifier_dropout	null
directionality	"bidi"
hidden_act	"gelu"
hidden_dropout_prob	0.1
hidden_size	768
initializer_range	0.02
intermediate_size	3072
layer_norm_eps	1e-12
max_position_embeddings	512
model_type	"bert"
num_attention_heads	12
num_hidden_layers	12
pad_token_id	0
pooler_fc_size	768
pooler_num_attention_heads	12
pooler_num_fc_layers	3
pooler_size_per_head	128
pooler_type	"first_token_transform"
position_embedding_type	"absolute"

torch_dtype	"float32"
transformers_version	"4.17.0"
type_vocab_size	2
use_cache	true
vocab_size	21128

而以下爲我在 question answering 之中所使用的 BERT 的 configuration：

_name_or_path	"bert-base-chinese"
architectures	["BertForQuestionAnswering"]
attention_probs_dropout_prob	0.1
classifier_dropout	null
directionality	"bidi"
hidden_act	"gelu"
hidden_dropout_prob	0.1
hidden_size	768
initializer_range	0.02
intermediate_size	3072
layer_norm_eps	1e-12
max_position_embeddings	512
model_type	"bert"
num_attention_heads	12
num_hidden_layers	12
pad_token_id	0
pooler_fc_size	768
pooler_num_attention_heads	12
pooler_num_fc_layers	3
pooler_size_per_head	128
pooler_type	"first_token_transform"
position_embedding_type	"absolute"
torch_dtype	"float32"
transformers_version	"4.17.0"
type_vocab_size	2
use_cache	true
vocab_size	21128

在 multiple choice 和 question answering 之中，我皆使用了 AdamW 作爲 optimizer，以及使用 transformers 所提供的 get_linear_schedule_with_warmup 作爲 learning rate scheduler，並且，我使用了 cross entropy 作爲 loss function，其中，我使用的 batch size 爲 16，learning rate 爲 0.00005，weight decay 爲 0.01，scheduler 的 warmup step 爲 total step 的 $\frac{1}{10}$，finetune 了 5 個 epoch，以此得到最終的 BERT model。我所訓練出來的 BERT 在 Kaggle public leaderboard 上所得到的 accuracy 爲 0.74773。

Problem 2-2

除了 BERT 之外，我於本次作業之中也有嘗試使用MacBERT。MacBERT 的 model 架構與 BERT 相同，兩者唯一的差別在於 pretraining task，MacBERT 會在中文資料上進行 pretrain，在 MLM task 之中，MacBERT 會使用 N-gram masking strategy 對 input sequence 進行 whole word masking，將 input sequence 中 15% 的 word 進行遮罩，但 MacBERT 並不會將這些 word 替換成 MASK token，而是會將這些要被遮罩的 word 之中，80% 使用 word2vec 找出與要被遮罩的 word 意思相近的 word 來進行替換，而若找不到意思相近的 word 則會隨機選出一個 word 來進行替換，而在要被遮罩的 word 之中有 10% 會使用隨機的 word 來進行替換，最後剩下的 10% 則會用原本的 word 來進行遮罩，以此方式來訓練 MLM task，除了 MLM task 之外，MacBERT 還使用了 SOP task 來進行 pretrain，而不是使用 BERT 的 NSP task，以此來得到最後的 pretrained model。

以下爲我在 multiple choice 之中所使用的 MacBERT 的 configuration：

_name_or_path	"hfl/chinese-macbert-base"
architectures	["BertForMultipleChoice"]
attention_probs_dropout_prob	0.1
classifier_dropout	null
directionality	"bidi"
gradient_checkpointing	false
hidden_act	"gelu"
hidden_dropout_prob	0.1
hidden_size	768
initializer_range	0.02
intermediate_size	3072
layer_norm_eps	1e-12
max_position_embeddings	512
model_type	"bert"
num_attention_heads	16
num_hidden_layers	12
pad_token_id	0
pooler_fc_size	768
pooler_num_attention_heads	12
pooler_num_fc_layers	3
pooler_size_per_head	128
pooler_type	"first_token_transform"
position_embedding_type	"absolute"
torch_dtype	"float32"
transformers_version	"4.17.0"
type_vocab_size	2
use_cache	true
vocab_size	21128

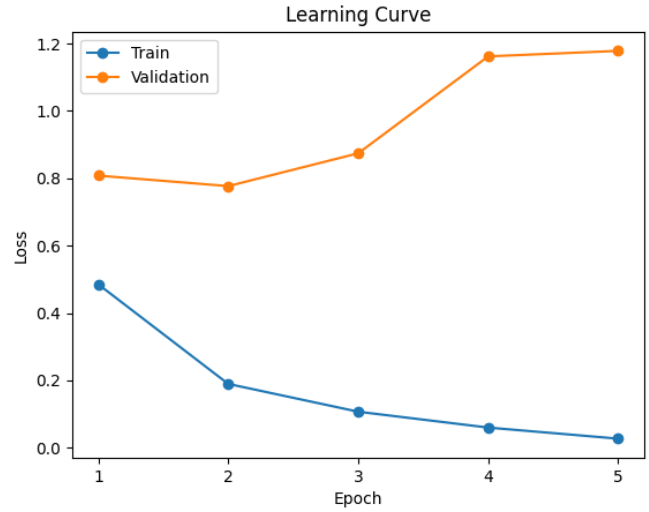
而以下爲我在 question answering 之中所使用的 BERT 的 configuration：

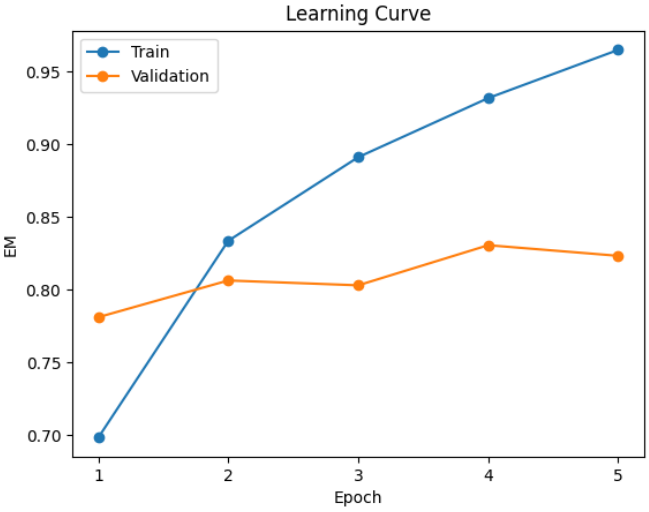
_name_or_path	”hfl/chinese-macbert-large”
architectures	[”BertForQuestionAnswering”]
attention_probs_dropout_prob	0.1
classifier_dropout	null
directionality	”bidi”
gradient_checkpointing	false
hidden_act	”gelu”
hidden_dropout_prob	0.1
hidden_size	1024
initializer_range	0.02
intermediate_size	4096
layer_norm_eps	1e-12
max_position_embeddings	512
model_type	”bert”
num_attention_heads	16
num_hidden_layers	24
pad_token_id	0
pooler_fc_size	768
pooler_num_attention_heads	12
pooler_num_fc_layers	3
pooler_size_per_head	128
pooler_type	”first_token_transform”
position_embedding_type	”absolute”
torch_dtype	”float32”
transformers_version	”4.17.0”
type_vocab_size	2
use_cache	true
vocab_size	21128

在 multiple choice 和 question answering 之中，我皆使用了 AdamW 作爲 optimizer，以及使用 transformers 所提供的 get_linear_schedule_with_warmup 作爲 learning rate scheduler，並且，我使用了 cross entropy 作爲 loss function，其中，我使用的 batch size 爲 16，learning rate 爲 0.00005，weight decay 爲 0.01，scheduler 的 warmup step 爲 total step 的 $\frac{1}{10}$，finetune 了 5 個 epoch，以此得到最終的 MacBERT model。此外，我於 multiple choice 和 question answering 之中皆訓練了兩個 MacBERT，並將兩個 model 所輸出的 logit 相加以此進行 ensemble，我所訓練出來的 MacBERT 在 Kaggle public leaderboard 上所得到的 accuracy 爲 0.80470。

Problem 3

下圖爲 MacBERT 在訓練時所得到的 learning curve：





Problem 4

以下爲我在 multiple choice 之中所使用的 BERT 的 configuration：

_name_or_path	”bert-base-chinese”
architectures	[”BertForMultipleChoice”]
attention_probs_dropout_prob	0.1
classifier_dropout	null
directionality	”bidi”
hidden_act	”gelu”
hidden_dropout_prob	0.1
hidden_size	768
initializer_range	0.02
intermediate_size	3072
layer_norm_eps	1e-12
max_position_embeddings	512
model_type	”bert”
num_attention_heads	12
num_hidden_layers	12
pad_token_id	0
pooler_fc_size	768
pooler_num_attention_heads	12
pooler_num_fc_layers	3
pooler_size_per_head	128
pooler_type	”first_token_transform”
position_embedding_type	”absolute”
torch_dtype	”float32”
transformers_version	”4.17.0”
type_vocab_size	2
use_cache	true
vocab_size	21128

而以下爲我在 question answering 之中所使用的 BERT 的 configuration：

_name_or_path	”bert-base-chinese”
architectures	[”BertForQuestionAnswering”]
attention_probs_dropout_prob	0.1
classifier_dropout	null
directionality	”bidi”
hidden_act	”gelu”
hidden_dropout_prob	0.1
hidden_size	768
initializer_range	0.02
intermediate_size	3072
layer_norm_eps	1e-12
max_position_embeddings	512
model_type	”bert”
num_attention_heads	12
num_hidden_layers	12

pad_token_id	0
pooler_fc_size	768
pooler_num_attention_heads	12
pooler_num_fc_layers	3
pooler_size_per_head	128
pooler_type	"first_token_transform"
position_embedding_type	"absolute"
torch_dtype	"float32"
transformers_version	"4.17.0"
type_vocab_size	2
use_cache	true
vocab_size	21128

在 multiple choice 和 question answering 之中，我皆使用了 AdamW 作爲 optimizer，以及使用 transformers 所提供的 get_linear_schedule_with_warmup 作爲 learning rate scheduler，並且，我使用了 cross entropy 作爲 loss function，其中，我使用的 batch size 爲 16，learning rate 爲 0.00005，weight decay 爲 0.01，scheduler 的 warmup step 爲 total step 的 $\frac{1}{10}$，訓練了 5 個 epoch。我使用 pre-trained model 所訓練出來的 BERT，其在 multiple choice 和 question answering 之中所得到的 validation accuracy 分別爲 0.97 和 0.83，但從頭重新訓練所得到的 BERT，其在 multiple choice 之中所得到的 validation accuracy 僅有 0.53，而在 question answering 之中則是完全訓練不起來，我也有根據助教所提供的提示，試著將模型的參數減少來訓練，也有試過調整 hyper-parameter，但是所得到的結果並沒有改善。

Problem 5

以下爲我在 intent classification 之中所使用的 BERT 的 configuration：

_name_or_path	"bert-base-uncased"
architectures	["BertForSequenceClassification"]
attention_probs_dropout_prob	0.1
classifier_dropout	null
gradient_checkpointing	false
hidden_act	"gelu"
hidden_dropout_prob	0.1
hidden_size	768
initializer_range	0.02
intermediate_size	3072
layer_norm_eps	1e-12
max_position_embeddings	512
model_type	"bert"
num_attention_heads	12
num_hidden_layers	12
pad_token_id	0

position_embedding_type	"absolute"
torch_dtype	"float32"
transformers_version	"4.17.0"
type_vocab_size	2
use_cache	true
vocab_size	30522

在 intent classification 之中，我會將輸入的句子進行 tokenization，接著在該句子的頭尾分別加上 CLS token 和 SEP token，再將其長度 pad 到 512，以此進行 data preprocessing，而在訓練時，我使用了 AdamW 作爲 optimizer，以及使用 transformers 所提供的 get_linear_schedule_with_warmup 作爲 learning rate scheduler，並且，我使用了 cross entropy 作爲 loss function，其中，我使用的 batch size 爲 8，learning rate 爲 0.00005，weight decay 爲 0.01，scheduler 的 warmup step 爲 total step 的 $\frac{1}{10}$，finetune 了 5 個 epoch，以此得到最終的 BERT model。我於作業一之中從頭訓練了的一個 Transformer (使用了 PyTorch 的 TransformerEncoder)，其在 Kaggle public leaderboard 上所得到的 accuracy 爲 0.93777，而在改爲使用 pretrained 的 BERT 之後，accuracy 上升到了 0.96666。

而以下爲我在 slot tagging 之中所使用的 BERT 的 configuration：

_name_or_path	"bert-base-uncased"
architectures	["BertForTokenClassification"]
attention_probs_dropout_prob	0.1
classifier_dropout	null
gradient_checkpointing	false
hidden_act	"gelu"
hidden_dropout_prob	0.1
hidden_size	768
initializer_range	0.02
intermediate_size	3072
layer_norm_eps	1e-12
max_position_embeddings	512
model_type	"bert"
num_attention_heads	12
num_hidden_layers	12
pad_token_id	0
position_embedding_type	"absolute"
torch_dtype	"float32"
transformers_version	"4.17.0"
type_vocab_size	2
use_cache	true
vocab_size	30522

在 intent classification 之中，我會將輸入句子中的每個 word 進行 tokenization，並將每個 word 的 tag 標記在其第一個 token 上，而該 word 的其它 token

則會 label 成-100 (根據 Hugging Face 的官方文件所述，當一個位置的 label 爲-100 時，則不會將該位置的輸出用來計算 loss)，接著在該句子的頭尾分別加上 CLS token 和 SEP token，再將其長度 pad 到 512，以此進行 data preprocessing，而在訓練時，我使用了 AdamW 作爲 optimizer，以及使用 transformers 所提供的 get_linear_schedule_with_warmup 作爲 learning rate scheduler，並且，我使用了 cross entropy 作爲 loss function，其中，我使用的 batch size 爲 8，learning rate 爲 0.00005，weight decay 爲 0.01，scheduler 的 warmup step 爲 total step 的 $\frac{1}{10}$，finetune 了 5 個 epoch，以此得到最終的 BERT model。我於作業一之中訓練出來的 LSTM，其在 Kaggle public leaderboard 上所得到的 accuracy 爲 0.80536，而在改爲使用 pretrained 的 BERT 之後，accuracy 上升到了 0.82949。